DESIGN OF A META-ASSEMBLER

By

SHARADCHANDRA R. MURTHY

Bachelor of Engineering

Bangalore University

Bangalore, India

1984

Submitted to the faculty of the
Graduate college of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1988

Thesis
1988
M984d
cop. 2

DESIGN OF A META-ASSEMBLER

Thesis Approved:

Thesis Adviser

Dean of the Graduate college

1311830

ACKNWOLEDGMENTS

I wish to express my sincere appreciation to Dr. L.G.Johnson for his encouragement and advise throughout my graduate program. Thanks also to Dr. H.R.Bilger and Dr. R.L.Cummins and Dr. C.D.Latino for serving on my graduate committee.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. WHAT IS MICROPROGRAMMING ? 3

III. DESIGN OF A META-ASSEMBLER 7

Problem specification 7
Design 8
Data structures 11
Search strategy 23
Modularity 25

IV. CONCLUSION 28

APPENDIX A - MANUAL 29

APPENDIX B - SYNTAX 32

APPENDIX C - ERROR MESSAGES 38

APPENDIX D - SAMPLE MICROPROGRAM..................... 42

APPENDIX E - META-ASSEMBLER LISTINGS 45

APPENDIX F - SOURCE PROGRAM LISTING 53

LIST OF FIGURES

Figure | Page

1. Data structure for address table 14

2. Data structure for field table 16

3. Data structure for format table 19

4. Data structure for the microword 21

CHAPTER I

INTRODUCTION

A significant trend in computer design in recent years has been the replacement of the conventional transistor logic control section of a digital computer with "stored logic" or microprogrammed control, stored in high speed, nondestructive read only storage.

The main reason for this shift in computer control implementation is the economic superiority of microprogramming over conventional logic control. Microprogramming has made it economically feasible, for example, to have the same comprehensive instruction set built into a whole line of new computers, even the smallest ones. Thus we have computer lines that have compatible instruction sets, yet their internal hardware, organization, and structure are drastically different. Microprogrammed control offers many other advantages besides the much publicized upwards and downwards compatibility. Among these being inherent flexibility of a microprogrammed system to permit architectural extensions and modifications that will make the system perform a specific data processing function with maximum efficiency.

All present indicators point to an almost explosive increase in microprogramming activity by system engineers

and users. The biggest factor that will lead to the projected increased exploitation of microprogrammed control is the inclusion of a non-destructive writable control storage. This would remove the restriction of permanent fixed operation codes locked in to the system architecture, and it would lead to a number of fascinating possibilities. Thus microprogrammed control provides a straightforward way to correct errors or provide enhanced capabilities in the instruction set.

CHAPTER II

WHAT IS MICROPROGRAMMING ?

A modern digital computer can be partitioned into five distinct functional units viz, input, storage, arithmetic and logic unit, output and control. These five units communicate with each other through electronic signals that represent data, instructions and control signals. The order timing and direction in which this information flows within and between the five principal functional sections in a computer are effected by the control unit, which in turn is directed by the sequence of machine instructions.

The control section of a computer directs the operation of the entire computer. It receives units of information from the storage section which tell it what operations are to be performed and where the data to be operated upon are located in the storage section. After the control section determines the exact instruction to be executed, it then issues control signals to open and close the specific gates throughout the system, thus permitting the necessary data in the form of electrical signals to flow from one functional unit to another in the execution of that operation. Once the ALU has finished with its part, the control section may issue the necessary control to permit the results to be transmitted back into the storage unit or

some output unit.

The general nature of the control unit is fourfold: the first two consisting of fetching and decoding of the sequence of machine instructions; the third function is the gating of the data paths to perform operations on the data fields; and the fourth is changing the state of the computer so as to allow the next required operation to be performed.

Thus a digital computer may simply be described as an elaborate array of logic network called data paths, which consist of static functions like adders, shifters, indicators, registers, parity circuits, mask circuits and other boolean functions. These static functions are interconnected by data busses which permit the information flow from one functional unit to another. All these data paths are static in nature and can be activated by the enabling and disabling signals which emanate from the control section. The control signals are made up of clock pulses for timing, decoding, sequencing, and decision logic and they direct and control the operation of the total system over any number of consecutive clock cycles.

Traditionally this information was permanently built into the system by connecting a set of decoders and flip flops in an ad hoc manner which can be viewed as a tree of storage elements and signal wires tied together in an unsystematic way. A decision to make the slightest modification to the instruction set could entail a major modifi-

cation to the whole structure of the tree.

Microprogramming was originally conceived as an alternative design procedure to the ad hoc procedure applied to conventional hardware. It has eventually become an alternate design and implementation tool for the control section, where the hardware control is replaced by a stored logic section, or "microprogram control" section, stored in a high speed, nondestructive read only storage. The information stored therein is designed to control each function for each consecutive machine cycle.

Microprogramming, interpreted as implementing control logic, primarily by read only storage, cuts across the specialities of electronic module design, mechanical languages, programming, and systems architecture. It is therefore a promising means for designing integrated hardware-software systems. This method of control from the programmers point of view, is similar to writing a program in which a given arithmetic or logic operation is executed by giving the cpu or the system a step-by-step description of the job to be done. This program, then, is a series of subcommands for the functions built into the system.

Microprogramming is analogous to conventional programming. The user programmer tells the system what to do by placing instructions in the high speed main storage. The microprogrammer tells the system how to do it by controlling which storage and logic elements are used and how they are used for each operation. Thus the machine instruction

which the programmer considered to be the lowest level of communication with the system can now be viewed as a closed subroutine broken down into a sequence of more elementary functions called microinstructions. Each microinstruction is designed to specify the control gates that are opened at a particular point during the machine cycle.

Thus the microprogrammer essentially writes microcode in terms of logical 1's and 0's, which is a very error prone and tedious process. This makes programming very difficult at such low levels especially when the microwords are very long. It is a lot easier to code in a symbolic language rather than 1's and 0's of microcode just as programmers find it easier to code in a symbolic assembly language rather than the 1's and 0's of machine language. This symbolic language for microprogramming is called a "meta-assembler".

CHAPTER III

DESIGN OF A META-ASSEMBLER

The design of the meta-assembler will be discussed through a detailed presentation of the following topics to be covered in this chapter:

1. Problem specification

2. Design

3. Data structure

4. Search strategy and

5. Modularity

Problem Specification

The problem was essentially to develop a highly generic assembler which would allow the user to assemble microprograms with word lengths varying from 1 to 256 bits. A special syntax was to be developed for the user to communicate with the assembler. This syntax would contain certain pseudo instructions unique to the meta-assembler. The meta-assembler would be capable of recognizing the design format (Horizontal or Vertical) and would be as user friendly as possible and would be highly portable which

means the program would be capable of running on different hardware. The meta-assembler would run both in the MSDOS and UNIX environments.

Design

Since the program was to be highly portable, the C programming language was adopted to develop the assembler. The other reason for choosing C was the flexibility provided by C in allowing the programmer to define his/her own data structures. Also modular programming and top down design strategies were kept in mind for which C is so very conducive.

Since the microinstructions can be specified in single format or multiple formats, the program had to be capable of recognizing the format specification. For this a pseudoinstruction "FORMAT" has been used to tell the assembler that a particular statement in the microprogram is a statement specifying the format the user has adopted in his design methodology. A counter keeps track of the number of format statements and based on the number of format statements, the program creates format tables, containing the names of the fields present in each format statement. In addition to the names the table also contains information regarding the bit positions occupied by the respective fields in the microword. Not more than five format statements are allowed because of memory limitations.

After reading the format statements the assembler

expects the microprogrammer to define the symbols representing values in each field. For the assembler to recognize that the following data are the symbol names and their respective definitions, a pseudoinstruction "DEF" must precede the define statements. Once this pseudoinstruction is encountered, the assembler will know that the following text defines the symbols. The format in which the symbols are to be specified can be understood by looking up the section on syntax. The symbols are stored in tables to be referred to as field tables. The field tables are organized in such a manner that they contain the symbol names, their definitions, their position of origin in the microword and their size in bits. A more detailed picture of the field tables can be had in the section on data structures. The assembler keeps reading the symbols till it encounters a pseudoinstruction called "ENDEF" which tells the assembler that the definitions of symbols are over.

This meta-assembler has been designed to be a two pass assembler. The first pass is an address generation phase, where the assembler scans through the entire microprogram and locates address labels and assigns address values to these labels and in the process stores both these data in a table referred to as the address table.

In the second pass the meta-assembler scans through the entire microprogram to read the microinstructions. As each symbol specifying code for a particular field is encountered while scanning a microinstruction a check is

first made to see if the symbol is a field name, for example a branch address field. The assembler does this by looking for a reserved symbol '=' . All characters to the left of this symbol are grouped together to form a string which is supposed to be one of the field names. All the characters to the right of the '=' symbol must be a number or are assumed to represent the field value or an address label. If the string to the right of the '=' symbol is a number, then all the bits in the number are examined and the bits in the field space are assembled one after the other. If not a number, then a check of the address table is made to find the address value of the address label encountered. When a match is found the address value is read into a buffer and every bit in the two byte address value is checked and accordingly a bit in the address space of the microword is set. If not an address label, a check is made to see if it is a field symbol value by checking the field table. Then the field symbol value is examined bit by bit and accordingly assembled by setting the bits in the field space of the microword.

If the symbol does not specify a field name, then the meta-assembler searches for the field value symbol in the field table. A detailed description of the search technique can be found in the section on search strategy. Once the symbol is located in the field table, the assembler reads the position of origin of the symbol in the microword, the size of the symbol and the definition of the sym-

bol from the field table. Next the meta-assembler proceeds with the assembly process by examining the symbol definition bit by bit and correspondingly sets the bits in the microword starting from the position of origin of the symbol in the microword. This process is repeated till all the bits belonging to a particular symbol have been examined and accordingly assembled. The same process repeats for all symbols in all the microinstructions. The assembler terminates the assembly process on encountering a pseudo instruction called "END" which indicates the end of the microprogram or on encountering the end of file character.

Data Structures

Before getting into a discussion on the data structures designed for the meta-assembler, it is important to discuss about the structure that C permits according to Kernighan and Ritchie. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. The following example shows how a structure is declared in C and is the same structure that has been used to define the format table to be discussed later.

```
struct formatb {

     char formnam[8];

     int bitsize,posn;
```

```
}formatab[40];
```

This example declares an array formatab which is a structure of type formatb. This structure contains information pertaining to the fields in a format statement. The elements or variables mentioned in a structure are called members. In the example above formnam[8], bitsize and posn are all members of the structure formatb. The array contains 40 such structures. A member of a particular structure is referred to in an expression by a construction of the form

```
struct-name.member
```

The structure member operator "." connects the structure name and the member name. For the example under consideration, the following expression can be used for example to refer to the member formnam[j]

```
formatab[k].formnam[j]
```

There are a lot of data structures that have been specially designed for use in this program. These data structures have been organized for faster execution of the program. The organization of the data structures used will be discussed in detail in this section.

As discussed in the previous section the address labels and their corresponding values are stored in a table referred to as the address table. This table is a data

structure with name tag "addrtable" . This structure has been defined as shown below.

```
struct addrtable {
     char addrname[7];
     int contents;
}adtab[200];
```

As can be seen an array adtab has been declared to be of type addrtable. The members of the structure are addrname[7] where the names of the address label will be stored, and contents, where the absolute value assigned to that address value will be stored. Thus in order to find the adress value of a particular address label, one needs to search through the array and try to find a match with one of the addrname variables in the array. and when a match is found the contents variable pointed to by the same index gives the absolute address of that address label. A good picture of the organization of this data structure is shown in Fig 1. This data structure occupies two Kbytes of memory.

There is another table referred to as field table which holds the symbol names and related information like symbol definition, symbol size in bits and position of origin of the symbol in the microword. This table is a data structure with name tag "fieldtable". This structure has been defined as shown below.

ADDRESS TABLE

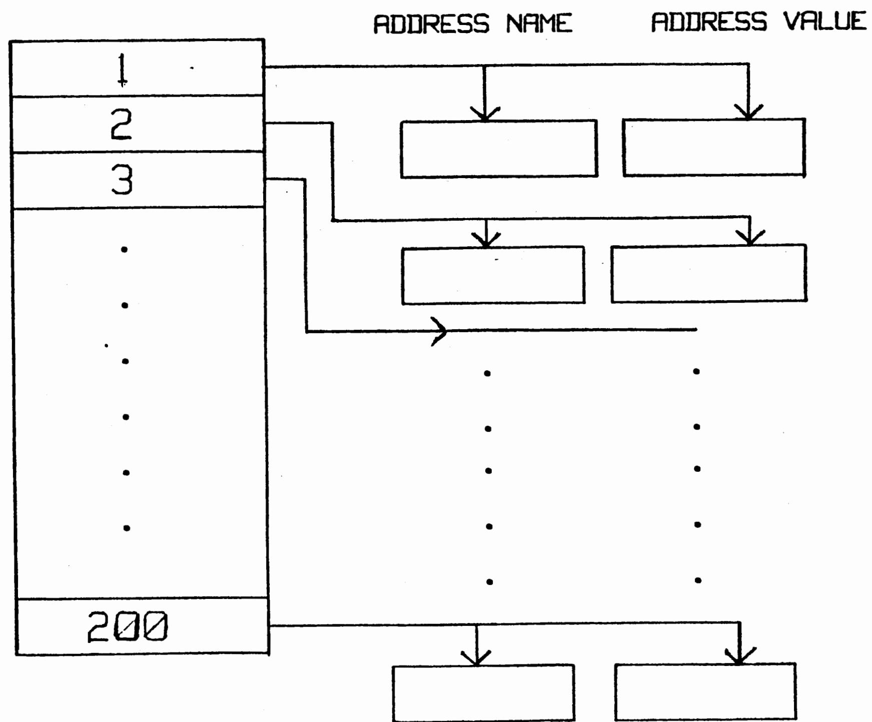

Figure 1. Data structure for address table.

```
struct fieldtable {

     struct symtable symtab[30];

     int tab_count;

}fieldtab[26];

struct symtable {

     char name[6],def[9];

     int bitlength,position;

};
```

This particular data structure is a little complex as a structure has been declared within a structure. The idea is to group all the information pertaining to a symbol in the field in a structure referred to as symtable. The members of this structure are the variables name[6] which holds the name of the symbol in the field, def[9] which holds the symbol value, bitlength which holds the size of the symbol in the field which is also the field size, and position which holds information regarding the position of origin of the symbol in the microword. Thus all the information pertaining to a particular symbol in a field can be retrieved from this structure. The array symtab[30] has been declared to be of type symtable, which means that 30 such structures can be stored in the array symtab. Now this data structure which will henceforth be called symbol table

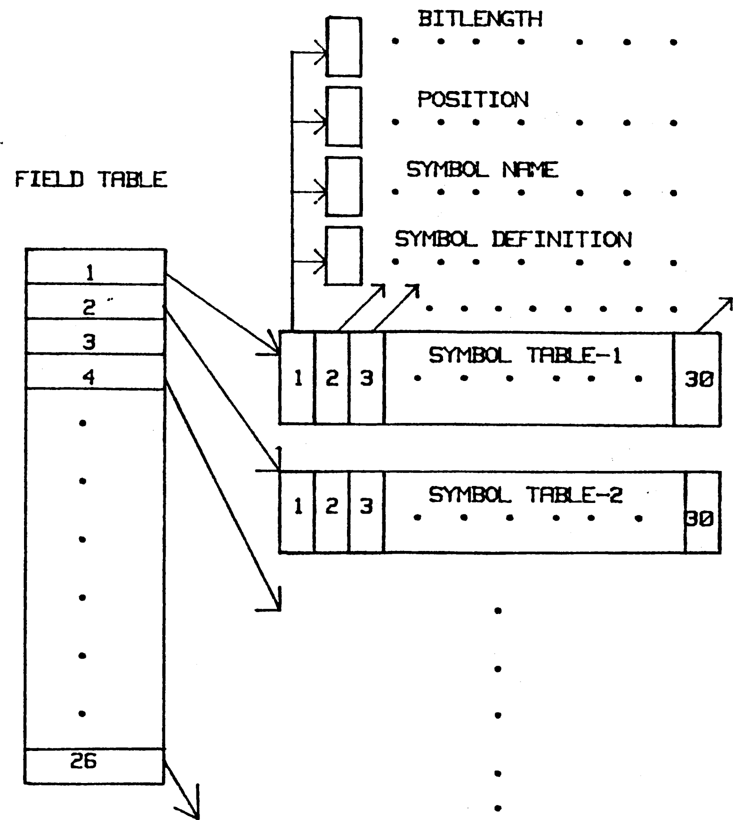

Figure 2. Data structure for field table.

is a member of another structure to be referred to as fieldtable. The other member of the fieldtable is tab_count which maintains a count of the locations occupied in the array symtab. The array fieldtab[26] has been declared to be of type fieldtable. Thus this data structure forms 26 symbol tables. Each symbol table can hold information about 30 symbols. This data structure was designed with the idea that each symbol table will contain symbols that have their names beginning with a particular English alphabetic character which has to be a capital letter. Thus the 26 symbol tables correspond to the 26 English alphabetic characters. A good picture of this data structure can be had by looking at Fig 2. This data structure occupies 15.6 Kbytes of memory. It will be a lot easier to appreciate the use of this data structure after looking at the section on search strategy where a detailed explanation of how this data structure aids in reducing the search time has been discussed. As can be seen the current data structure limits the number of symbols with symbol names having their first character corresponding to a particular capital letter to 30. In the future should a need arise to make room for more than 30 symbols the size of the symtab array must be increased from 30 to the desired value.

There is another table referred to as format table which holds the field names in every format statement and in addition to this also holds information regarding the position of origin of the field in the microword and the

size of each field in bits. The organization of this data structure can be seen in Fig 3. The structure with name tag forms defines this data structure. This structure is as shown below.

```
struct forms {

     struct formatb formatab[40];

}tab[5];

struct formatb {

     char formnam[8];

     int bitsize,posn;

}formatab[40];
```

The array formatab[40] has been declared to be of type formatb. Formatb is a structure which has 3 members viz, formnam[8] which holds the field name, bitsize which holds the size of the field and posn which holds the position of origin of the field in the microword. Thus this structure contains all the information pertaining to a field in a format statement. The array formatab contains 40 such structures. This means that information regarding 40 fields can be stored in this array. Now this data structure is a member of another structure with name tag forms. The array tab has been declared to be of type forms. This array is of size 5 which means that each element of this array

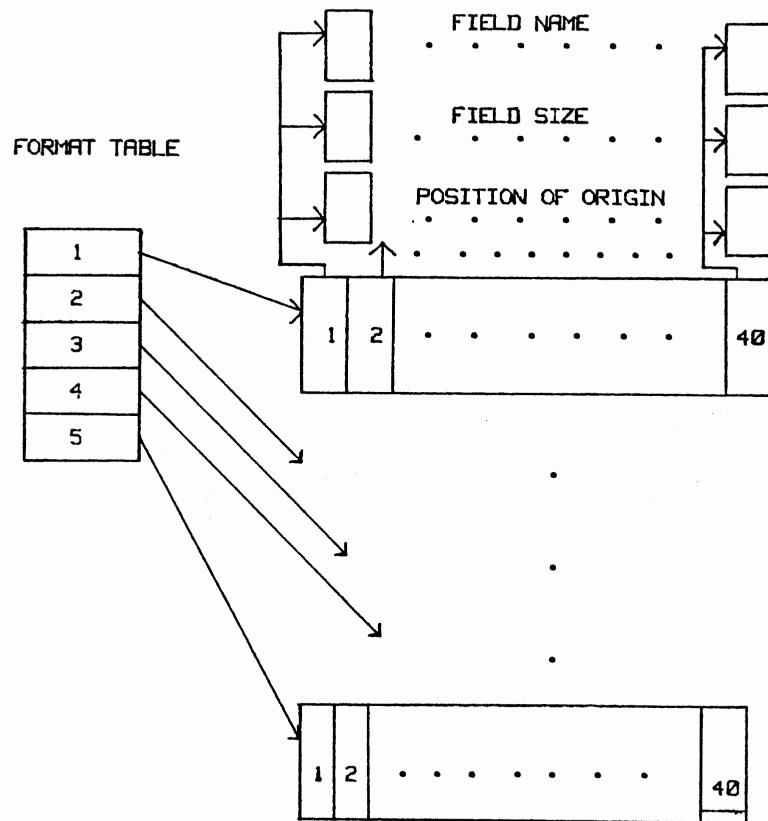

Figure 3. Data structure for format table.

corresponds to a format statement which can contain up to 40 fields. The five elements correspond to the five format statements thus limiting the number of format statements to five at present. This has been done because of memory limitations on the IBM pc. In the future, if one wants to make room for more than 5 format statements, all that needs to be done is to increase the size of the array tab to the desired value. This can be done only when the program runs in an UNIX environment. This data structure helps in identifying overlapping bitfields in a format statement. This data structure occupies two Kbytes of memory space.

The last data structure to be discussed in this section is a kind of data structure unique to the C language. A thorough understanding of this data structure will help a great deal in understanding the working of the meta-assembler. As the microwords have sizes varying from 1 to 256 bits which means a microword can span from 1 byte to 32 bytes. As the assembly process involves assembling bitfields whose sizes vary from 1 to 256 bits, it might be necessary that the assembler gain access to individual bits in the microword or gain access to one of the bytes that form the microword. The microwords are formed by chaining byte words. The chain has 1 to 32 bytes chained in series depending on the length of the microword. In the structure defined in the program which is also shown below, a union of a character variable over a data structure consisting of 8 bits is achieved.

Data structure for microword

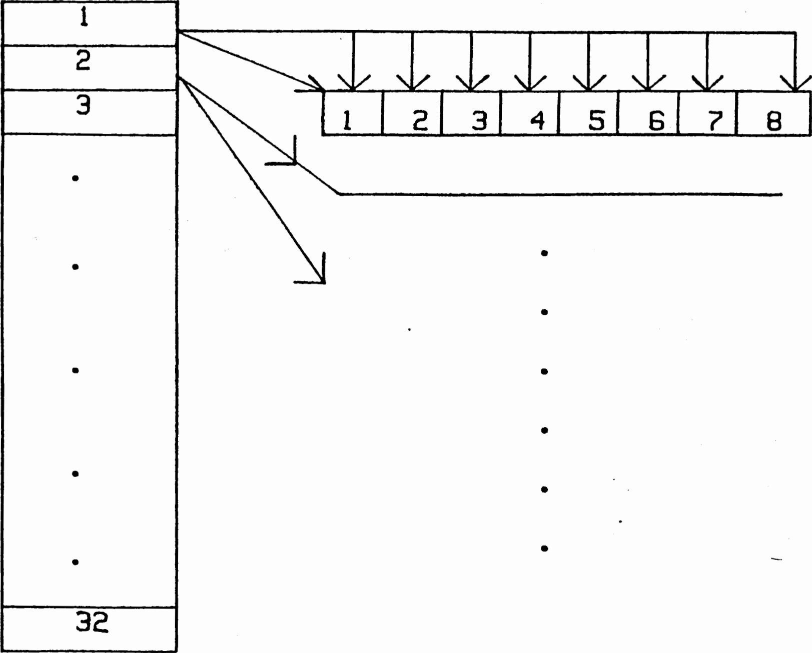

Figure 4. Data structure for microword.

```
union format {
    unsigned char all;
        struct {
            unsigned bit0:1;
            unsigned bit1:1;
            unsigned bit2:1;
            unsigned bit3:1;
            unsigned bit4:1;
            unsigned bit5:1;
            unsigned bit6:1;
            unsigned bit7:1;
        } part;
    }flag[32];
```

This kind of data structure is unique to the C language. This allows the byte to be accessed fully as a character or individual bits in the byte could be accessed and operations performed on the bits. Most of the operations involve setting the bits to either 1 or 0. The value '32' refers to the size of the microword in bytes. For the current problem it was desired that the maximum size of the microword be 256 bits which is 32 bytes. A microword of bigger size can be performed by simply changing this value from 32 to the desired value. No other alterations in the program are required. This shows how flexible the assembler becomes because of this data structure. Fig 4. shows the organization of this data structure. As can be seen the structure consists of a table of 32 pointers. Each pointer

points to a character variable which is a byte long. The pointer also points to each individual bit in this byte. To access a particular bit, the bit needs to be addressed. This is done as follows :

```
flag[10].part.bit0
```

This statement shows how the first bit of the 10th byte in the microword can be accessed. This data structure occupies 32 bytes of memory space.

Search Strategy

The meta-assembler spends seventy percent of its time in searching through the address, field and format tables. Most of the search involves the field table where the symbols and their related information are stored.

There were a lot of search methodologies that were considered before settling for the current technique. The linear search was considered first with a kind of data structure much different from the one shown in the earlier section on data structures. The binary search technique was considered next. Both these techniques did make programming very convenient but at the same time slowed down the execution time of the program considerably. This was highly unsuitable for the kind of problem in hand.

The other option was to go in for a hash table search which is a proven technique and is extremely fast. For the problem in hand, it was found that the currently designed

data structure with the search technique to be discussed in this section, formed a powerful combination in increasing the speed of execution of the program.

The current technique is very similar to the hashing approach. This technique led to the development of the data structure with name tag the fieldtable. The idea was to organize data in such a manner that the data base to be searched was cut in size considerably. As was already discussed in the section on data structures, the field table has twenty six pointers, the pointer values ranging from one to twenty six. Each of these twenty six pointers point to a separate table containing thirty pointers and each of these thirty pointers point to four variables simultaneously.

During the process of searching, if information about a particular symbol is to be retrieved, the first character of the symbol name which is supposed to be a capital letter is read into a character buffer and the three most significant bits of this byte (ASCII representation of a character) are masked off. As the syntax does not allow symbol names to begin with any other character except capital letters, the resultant integer value then ranges from one to twenty six. This value is then used to address the field table where there are twenty six pointers. Thus one of the twenty six pointers is selected, which in turn selects one of the twenty six tables. Now the search is restricted to just thirty symbols. This part of the search

is just a simple linear search. Once the symbol is located, this very pointer can be used to gain access to the corresponding information regarding that particular symbol. In a linear search, on the average, one needs to lookup at least half the number of elements in the table, which in this case turns out to be fifteen. Thus a search which would have involved a maximum possible lookup of seven hundred and eighty symbols has now been reduced to just fifteen symbols. This is more than a justification for having adopted the current search technique and the data structures discussed in the earlier section.

An improvement of a factor of seven in the execution speed of the program was observed by switching the search strategy from binary search to the current technique which is a mix of both the hashing approach and the linear approach.

Modularity

Top down design strategy has been adopted for the design of the meta-assembler. The problem in hand can be broken down into five main modules which are as listed below.

1. Initialization of all tables
2. Loading of symbol tables
3. Checking of overlapping bitfields

4. Pass one of the meta-assembler

5. Pass two of the meta-assembler

The program basically consists of these five principal modules which in turn refer to some smaller modules. The main program calls these five modules in the same sequence as shown above.

During the initialization phase, a pattern "eos" is written into the symbol name variables of all the tables in the field table, into all the address name variables in the address table and into all the field name variables in the format table. Also all bits in the microword are set to zero.

In the second module, all the define statements are read from the text file and the symbol names and their corresponding definitions are read and copied into the field tables. Also in this process the size of the symbol in bits is computed and entered into the field table along with the location of the symbol in the microword. Before making entries into the field table, the format statements are read first and the field names in the format statements are copied into the format tables corresponding to the format statements. When the define statements are being read, the location of the field in the microword and the size of the field in bits are entered into the format table. Thus before this module completes execution, the format tables and the field tables are setup in memory.

The third module checks for overlapping bitfields in any format statement. It accesses the format tables and makes use of the information about the location of the fields in the microword and their size in bits. If an overlap is found, an error message is flashed on the screen and the program is terminated.

The fourth module is the first pass of the meta-assembler which is also the address generation phase. It is in this phase that all the address labels are assigned absolute values and this data is entered into the address table.

The fifth module is the second pass of the meta-assembler and it is in this phase that the assembly of the microcode takes place. This module reads the microprogram from the text file and then refers to the field and address tables to read the symbol definitions and the absolute values of the address labels respectively to assemble the microcode bit by bit. The assembled microcode is stored in a user specified file. Also the microword can be separated into n slices with each slice being a byte wide. Each byte wide slice is stored in different files specified by the user. This is done to down load the assembled microcode into the memory simulator which is part of a logic simulator currently under development at OSU or a PROM programmer.

CHAPTER IV

CONCLUSION

METASM has turned out to be a highly generic and a highly portable meta-assembler. It has been tested in both the MSDOS and UNIX environments and has been found to be executing without any problems. The program has been designed so that the assembled output is stored in a format which is the desired input format for a logic simulator under development at OSU. This provides the designer the facility of immediately testing his/her microprogram by down loading the assembled microprogram into a memory module in the simulator.

Though the meta-assembler has shown satisfactory results, there is a possibility that bugs might show up at a later stage when more users start using the assembler. The source program has been adequately documented and because of the modularity of the program patching bugs is not seen as a difficult proposition.

APPENDIX A

MANUAL

STARTUP :

To invoke the meta-assembler type "METASM" <CR>.

This causes the operating system to load the meta-assembler program and the current working directory is displayed on the screen. This allows the user to view all the file names in the current working directory. Next the program prompts the user to type the input file name by displaying the message "SPECIFY INPUT FILE NAME :" to which the user is supposed to respond by typing the file name where the microprogram resides. Next the program prompts the user to type the output file name where the assembled microprogram is supposed to be stored by displaying the message "SPECIFY OUTPUT FILE NAME :" to which the user is supposed to respond by typing the file name where he wants the assembled microprogram to be stored. While typing the output file name the user can specify a list option by typing '-l' after the file name. The following example will make this clear.

Specify output file name : prog1 .... if listing not desired.

Specify output file name : prog1-l .... if listing desired.

If the user desires a debug listing then the user needs to type

Specify output file name : prog1-ld

Once this procedure is completed, the program displays the following messages in case there are no errors in the microprogram :

"Please wait! creation of tables in progress.

Please wait! checking for overlapping bitfields.

Good you have no overlapping bitfields.

Next depending on the number of bytes in the microword, the program prompts the user that many times with the following message.

"Specify name of output file number #n :"

Thus if the microword contains 4 bytes, the program will prompt the user to provide four different file names one after the other.

If a meta-assembler listing is desired, then a list file by the name "METASM.LST" will be generated. If only the list option is specified then this file will contain only the listing without any debug messages. If a debug option is specified, then debug messages are inserted into the list file.

APPENDIX B

SYNTAX

As in any assembler this meta-assembler has its own syntax. The program is accepted by the meta-assembler in a particular format to be explained in this section and the program should contain certain pseudo instructions which are unique to the meta-assembler.

The following are the reserved words and symbols which the user is not supposed to use as symbol names or part of symbol names in his microprogram:

FIELD, END, FORMAT, ENDEF, DEF, eos, ':', ';', ',', '='

It is recommended that the user use only capital letters whenever an English alphabet is used in specifying field names or symbol names. The microprogram always starts with a format statement as shown below :

"FORMAT CC,BRA,ALU,DBSRC,DBST,AMUX,MDMUX;"

As can be seen the word "FORMAT" must be followed by a space and then by the field names. The field names must be separated by a comma. The end of format statement is indicated by a semicolon. No two fields in the same format statement can have the same bit positions in the microword. There might be occasions when a format statement might be very long and might span a few lines as shown below :

"FORMAT CC,BRA,DFERT,JHYDT,KJHSGY,FDGT,

GHDTY,GBDF,DGEYD,KHDHD,BHDH;"

The field names can not be more than 9 characters

long. The program is allowed a maximum of five format statements. The format statement must be followed by the "DEF" statement. It is this statement that tells the meta-assembler that the data following this statement are symbol definitions. The "DEF" statement is followed by a series of "FIELD" statements depending on the number of fields desired. Each field statement is followed by a series of symbol names and their corresponding definitions. The format for the field statements is as shown below.

```
"FIELD ALU 19,17;
ENALU=011;
ALJW=110;
FIELD BRA 27,20;
ADRSS=1111111111;"
```

The field statement must contain the pseudo instruction FIELD followed by a space and then the field name. The field names can contain any ASCII characters excluding the reserved words and symbols with the condition that the field name contains not more than 9 characters. The field name must be followed by a space and this must be followed by the bit position indicators. As can be seen the end position must be specified first followed by a separator which must be a comma followed by start position. This statement must be terminated by a semicolon as shown. The field statement is followed by one or more symbol definition statements. Each symbol definition statement must contain the field symbol name followed by a '=' character fol-

lowed by the field symbol value. The field symbol name must have one of the capital letters as the first character. The remaining characters can be any ASCII character. The field symbol names cannot be more than 9 characters long. The field symbol value cannot have more than 9 bits. Every field symbol definition statement must be terminated by a ';' character.

In the example shown above the field "ALU" occupies bits 17 to 19, that is a space of three bits. "ENALU" and "ALJW" are the two field symbols in the field "ALU".

The field statements must be followed by an ENDEF statement. This statement tells the assembler that all symbol definitions are complete and that the following text is the microprogram. The microprogram contains several lines of microinstructions. Each line of microinstruction may contain several field symbol names. All field symbol names must be separated by a ',' character. If there is an address label associated with that particular microinstruction, the label must precede the microinstruction. The address label must be terminated by a ':' character which is then followed by all the field symbol names belonging to that microinstruction. This is as shown below.

```
GETC: FUNC,ALKJ,BHGTY;
```

In the example shown above, GETC is the address label and the three field symbol names are FUNC, ALKJ, BHGTY. All field symbols are defined to be in the field in which they

are defined in the DEF section. If it is desired that a particular field symbol be used in more than one field, the '=' operator can be used to override the field definition by writing

```
FIELD NAME = FIELD SYMBOL NAME;
```

This can be used only if the size of the field to which the field symbol is being assigned is same as that of the field in which the field symbol has already been defined in the DEF section. For example

```
PUTC: EHJK,ALU=SMJK,HJYE;
```

In this example ALU is the field name to which the field symbol SMJK defined for another field, is being assigned. The '=' operator is also used to assign address label values or numbers to the field. To specify a branch address for example, the field name must be specified first followed by a '=' character followed by a numeric value or an address label. The syntax for specifying branch address is as shown below.

```
GETC: FUNC,BRA=PUTC;
REST:ENALU,BRA=09;


PUTC:......,.....;
```

Each microinstruction must be terminated by a ';' character. This character is recognized as the end of a microinstruction. The microprogram must end with an END

statement. The syntax for specifying branch address is as shown below.

A very good picture of the format for writing a microprogram can be had by looking at the sample micropro-gram attached to this report.

APPENDIX C

ERROR MESSAGES

1. When "FORMAT" statement is missing the following error message is displayed on the screen and the assembly process is terminated.

   "error : FORMAT statement missing "

   The likely cause of this can either be a spelling mistake or the characters might not be capital letters or the statement might be really missing.

2. If a space character does not follow the word "FORMAT" the following error message is displayed on the screen, but the meta-assembler can recover from this error. The user is advised to correct the error for proper documentation.

   "error: Type space after FORMAT"

3. When DEF statement is missing the following error message is displayed on the screen.

   "error: DEF statement missing"

   The cause of this might be a spelling mistake or the characters might not be capital letters or the statement might be really missing

4. When a space character is missing after the word "FIELD" in the field statement, the following message is displayed on the screen.

   "error: enter a space after FIELD"

   On seeing this message on the screen the user is advised to check all the field statements in his program.

5. When a field name is more than 9 characters long, the following error message is displayed and the assembly process is terminated.

   "error: field name more than 9 characters long"

   To locate the field name in error the user is advised to take a listing in which the field name will be visible. The cause of this error is a missing ',' or ';' in the format statements. Check your format statements.

6. When the first character of a symbol name does not start with a capital letter, the following error message is displayed.

   "error: symbol name does not begin with a capital

letter"

"symbol :xxxxx"

Following the statement containing the error message is a statement containing the symbol which is in error

7. When any two fields in the same format statement overlap, the following error message is displayed on the screen.

   "error: You have overlapping bitfields in your definition"

   "FIELD:xxxxx overlaps FIELD:yyyyy"

   The user can identify the two overlapping bitfields by reading their names on the screen.

8. When a field is assigned a field symbol belonging to some other field, if the sizes of the two fields are not same, the following error message is displayed.

   "error: Field size does not match field symbol size"

   "Field :xxxxx"

   "Field symbol :yyyyy"

   " in -microinstruction "

9. When END statement is missing the following message is displayed.

   "error: END statement missing"

   The assembler can recover from this error.

10. When an address label is more than 6 characters long, the following message is displayed.

    "error: address label more than 6 characters long"

    Next the assembly process is terminated.

11. When the jump address specified in the microprogram does not match the address labels specified the following message is displayed.

    "error: address label xxxxxx not found"

    Next the program is terminated.

12. When a symbol encountered in the microinstruction does

not exist according to the symbol definition statements, the following message is displayed

"error: symbol not found"

"symbol :xxxx in"

"microinstruction"

The user can easily locate the error.

13. When reading the definition statements, if the definition part contains any character other than a 1 or 0 an error message is flashed along with the text in the microprogram where the error occurred.

14. When ENDEF statement is missing the following error message is displayed on the screen.

    "error: field name too long "

APPENDIX D

SAMPLE MICROPROGRAM

```
FORMAT CC,BRA,ALU,DBSRC,DBDST,AMUX,BMUX,MDMUX,FLAG;
FORMAT CC,BRA,ALUSRC,ALUOP,ALUDST,DBSRC,DBDST,AMUX,
                                   BMUX,MDMUX,FLAG;
FORMAT CC,VAL,ALU,DBSRC,DBDST,AMUX,BMUX,MDMUX,FLAG;
FORMAT CC,A,B,ALUSRC,ALUOP,ALUDST,DBSRC,DBDST,AMUX,
                                   BMUX,MDMUX,FLAG;
DEF
FIELD CC 31,28;
CONT=0000;
JUMP=0001;
JSUB=0010;
RET=0011;
LDC=0100;
JIR=0101;
JAM=0110;
JN=0111;
JZ=1000;
JC=1001;
JRUN=1010;
JBOT=1011;
JRES=1100;
FIELD BRA 27,20;
FIELD VAL 27,20;
FIELD A 27,24;
APC=1111;
EA=1110;
OPAND=1101;
DIVISOR=1100;
RINDEX=0111;
FIELD B 23,20;
BPC=1111;
BEA=1110;
BOPAND=1101;
BDIVISOR=1100;
FIELD ALU 19,12;
ALNOP=00000000;
FIELD ALUSRC 19,17;
BA=001;
QO=010;
BO=011;
AO=100;
OD=111;
FIELD ALUOP 16,14;
PLUS=000;
MINUS=001;
COMP=010;
OR=011;
AND=100;
INC=101;
COMP1=110;
FIELD ALUDST 13,12;
TO_Q=00;
TO_B=01;
RSHBQ=10;
```

```
LSHBQ=11;
FIELD DBSRC 11,10;
ENALU=01;
MMRD=10;
IPRD=11;
FIELD DBDST 09,07;
LDAR=001;
LDIR=010;
OPWR=011;
MMWR=100;
FIELD AMUX 06,06;
A<-IR=1;
FIELD BMUX 05,04;
B<-IR=01;
B<-IREVEN=10;
B<-IRODD=11;
FIELD MDMUX 03,02;
MUL=01;
DIV=10;
FIELD FLAG 01,00;
LDFL=01;
CLFL=10;
ENDEF
START
REST: JRUN,BRA=FETC,ALNOP;
      BPC,BO,AND,TO_B;
STAR: JRUN,BRA=FETC,ALNOP;
      JRES,BRA=REST,ALNOP;
      JBOT,BRA=BOOT,ALNOP;
      JUMP,BRA=STAR,ALNOP;
BOOT: BPC,BO,OR,TO_B,ENALU,LDAR;
      BPC,BO,INC,TO_B,IPRD,MMWR;
      JUMP,BRA=STAR,ALNOP;
FETC: BPC,BO,OR,TO_B,MMRD,LDIR;
      BPC,BO,INC,TO_B,MMRD,LDIR;
      JIR,ALNOP;
TRA: JUMP,BRA=STAR,A<-IR,B<-IR,AO,OR,TO_B;
END
```

APPENDIX E

META-ASSEMBLER LISTINGS

META-ASSEMBLER LISTING WITH DEBUG OPTION

This meta-assembler has been designed by
SHARAD MURTHY
under the guidance of
Dr. L. G. JOHNSON

```
M E T A - A S S E M B L E R   L I S T I N G
-----------------------------------------------

INPUT FILENAME :johnson.c
OUTPUT FILENAME :sam
FORMAT CC,BRA,ALU,DBSRC,DBDST,AMUX,BMUX,MDMUX,FLAG;

CC
BRA
ALU
DBSRC
DBDST
AMUX
BMUX
MDMUX
FLAG
FORMAT CC,BRA,ALUSRC,ALUOP,ALUDST,DBSRC,DBDST,AMUX,
       BMUX,MDMUX,FLAG;

CC
BRA
ALUSRC
ALUOP
ALUDST
DBSRC
DBDST
AMUX
BMUX
MDMUX
FLAG
FORMAT CC,VAL,ALU,DBSRC,DBDST,AMUX,BMUX,MDMUX,FLAG;

CC
VAL
ALU
DBSRC
DBDST
AMUX
BMUX
MDMUX
FLAG
FORMAT CC,A,B,ALUSRC,ALUOP,ALUDST,DBSRC,DBDST,AMUX,
       BMUX,MDMUX,FLAG;

CC
A
B
```

```
ALUSRC
ALUOP
ALUDST
DBSRC
DBDST
AMUX
BMUX
MDMUX
FLAG
DEF

FIELD CC 31,28;
CONT=0000;
JUMP=0001;
JSUB=0010;
RET=0011;
LDC=0100;
JIR=0101;
JAM=0110;
JN=0111;
JZ=1000;
JC=1001;
JRUN=1010;
JBOT=1011;
JRES=1100;
FIELD BRA 27,20;
FIELD VAL 27,20;
FIELD A 27,24;
APC=1111;
EA=1110;
OPAND=1101;
DIVISOR=1100;
FIELD B 23,20;
BPC=1111;
BEA=1110;
BOPAND=1101;
BDIVISOR=1100;
FIELD ALU 19,12;
ALNOP=00000000;
FIELD ALUSRC 19,17;
BA=001;
QO=010;
BO=011;
AO=100;
OD=111;
FIELD ALUOP 16,14;
PLUS=000;
MINUS=001;
COMP=010;
OR=011;
AND=100;
INC=101;
COMP1=110;
FIELD ALUDST 13,12;
```

```
 TO_Q=00;
 TO_B=01;
 RSHBQ=10;
 LSHBQ=11;
 FIELD DBSRC 11,10;
 ENALU=01;
 MMRD=10;
 IPRD=11;
 FIELD DBDST 09,07;
 LDAR=001;
 LDIR=010;
 OPWR=011;
 MMWR=100;
 FIELD AMUX 06,06;
 A<-IR=1;
 FIELD BMUX 05,04;
 B<-IR=01;
 B<-IREVEN=10;
 B<-IRODD=11;
 FIELD MDMUX 03,02;
 MUL=01;
 DIV=10;
 FIELD FLAG 01,00;
 LDFL=01;
 CLFL=10;
 ENDEF
 word length=4 Please wait, checking for overlapping bit-
fields FORMAT CC,BRA,ALU,DBSRC,DBDST,AMUX,BMUX,MDMUX,FLAG;

START

REST = 0
      BPC,BO,AND,TO_B;

STAR: JRUN,BRA=FETC,ALNOP;

STAR = 2
      JRES,BRA=REST,ALNOP;

      JBOT,BRA=BOOT,ALNOP;

      JUMP,BRA=STAR,ALNOP;

BOOT: BPC,BO,OR,TO_B,ENALU,LDAR;

BOOT = 6
      BPC,BO,INC,TO_B,IPRD,MMWR;

      JUMP,BRA=STAR,ALNOP;

FETC: BPC,BO,OR,TO_B,MMRD,LDIR;

FETC = 9
      BPC,BO,INC,TO_B,MMRD,LDIR;
```

```
      JIR,ALNOP;

TRA: JUMP,BRA=STAR,A<-IR,B<-IR,AO,OR,TO_B;

TRA = c END

PASS-1 complete PASS-2 BEGINS ===> Assembly in progress

REST: JRUN,BRA=FETC,ALNOP; 0 : a0 90 0 0

      BPC,BO,AND,TO_B; 1 : 0 f7 10 0

STAR: JRUN,BRA=FETC,ALNOP; 2 : a0 90 0 0

      JRES,BRA=REST,ALNOP; 3 : c0 0 0 0

      JBOT,BRA=BOOT,ALNOP; 4 : b0 60 0 0

      JUMP,BRA=STAR,ALNOP; 5 : 10 20 0 0

BOOT: BPC,BO,OR,TO_B,ENALU,LDAR; 6 : 0 f6 d4 80

      BPC,BO,INC,TO_B,IPRD,MMWR; 7 : 0 f7 5e 0

      JUMP,BRA=STAR,ALNOP; 8 : 10 20 0 0

FETC: BPC,BO,OR,TO_B,MMRD,LDIR; 9 : 0 f6 d9 0

      BPC,BO,INC,TO_B,MMRD,LDIR; a : 0 f7 59 0

      JIR,ALNOP; b : 50 0 0 0

TRA: JUMP,BRA=STAR,A<-IR,B<-IR,AO,OR,TO_B; c : 10 28 d0 50

END
 Symbol table status :
  number of symbols starting with character  A = 5
  number of symbols starting with character  B = 9
  number of symbols starting with character  C = 4
  number of symbols starting with character  D = 2
  number of symbols starting with character  E = 2
  number of symbols starting with character  F = 0
  number of symbols starting with character  G = 0
  number of symbols starting with character  H = 0
  number of symbols starting with character  I = 2
  number of symbols starting with character  J = 10
  number of symbols starting with character  K = 0
  number of symbols starting with character  L = 5
  number of symbols starting with character  M = 4
  number of symbols starting with character  N = 0
  number of symbols starting with character  O = 4
  number of symbols starting with character  P = 1
  number of symbols starting with character  Q = 1
  number of symbols starting with character  R = 3
```

```
 number of symbols starting with character  S = 0
 number of symbols starting with character  T = 2
 number of symbols starting with character  U = 0
 number of symbols starting with character  V = 0
 number of symbols starting with character  W = 0
 number of symbols starting with character  X = 0
 number of symbols starting with character  Y = 0
 number of symbols starting with character  Z = 0
Total number of symbols used = 54
```

META-ASSEMBLER LISTING WITHOUT DEBUG OPTION

```
             This meta-assembler has been designed by
                          SHARAD  MURTHY
                      under the  guidance of
                        Dr. L. G. JOHNSON
          M E T A - A S S E M B L E R    L I S T I N G
          ----------------------------------------------
 INPUT FILENAME :johnson.c
 OUTPUT FILENAME :samn
 word length=4
 Please wait, checking for overlapping bitfields
 PASS-1 complete
 PASS-2 BEGINS ===> Assembly in progress

REST: JRUN,BRA=FETC,ALNOP; 0 : a0 90 0 0

      BPC,BO,AND,TO_B; 1 : 0 f7 10 0

STAR: JRUN,BRA=FETC,ALNOP; 2 : a0 90 0 0

      JRES,BRA=REST,ALNOP; 3 : c0 0 0 0

      JBOT,BRA=BOOT,ALNOP; 4 : b0 60 0 0

      JUMP,BRA=STAR,ALNOP; 5 : 10 20 0 0

BOOT: BPC,BO,OR,TO_B,ENALU,LDAR; 6 : 0 f6 d4 80

      BPC,BO,INC,TO_B,IPRD,MMWR; 7 : 0 f7 5e 0

      JUMP,BRA=STAR,ALNOP; 8 : 10 20 0 0

FETC: BPC,BO,OR,TO_B,MMRD,LDIR; 9 : 0 f6 d9 0

      BPC,BO,INC,TO_B,MMRD,LDIR; a : 0 f7 59 0

      JIR,ALNOP; b : 50 0 0 0

TRA: JUMP,BRA=STAR,A<-IR,B<-IR,AO,OR,TO_B; c : 10 28 d0 50

END Symbol table status :
  number of symbols starting with character  A = 5
  number of symbols starting with character  B = 9
  number of symbols starting with character  C = 4
  number of symbols starting with character  D = 2
  number of symbols starting with character  E = 2
  number of symbols starting with character  F = 0
  number of symbols starting with character  G = 0
  number of symbols starting with character  H = 0
  number of symbols starting with character  I = 2
  number of symbols starting with character  J = 10
  number of symbols starting with character  K = 0
```

```
 number of symbols starting with character  L = 5
 number of symbols starting with character  M = 4
 number of symbols starting with character  N = 0
 number of symbols starting with character  O = 4
 number of symbols starting with character  P = 1
 number of symbols starting with character  Q = 1
 number of symbols starting with character  R = 3
 number of symbols starting with character  S = 0
 number of symbols starting with character  T = 2
 number of symbols starting with character  U = 0
 number of symbols starting with character  V = 0
 number of symbols starting with character  W = 0
 number of symbols starting with character  X = 0
 number of symbols starting with character  Y = 0
 number of symbols starting with character  Z = 0
Total number of symbols used = 54
```

APPENDIX F

SOURCE CODE LISTING

```
/*************************************************************/
/*                                                           */
/*                   M E T A S M - 87                        */
/*                   ----------------                        */
/*                                                           */
/*                          A                                */
/*                                                           */
/*                    Meta-assembler                         */
/*                                                           */
/*                       designed                            */
/*                                                           */
/*                          by                               */
/*                                                           */
/*                   Sharad R Murthy                         */
/*                                                           */
/*               Advisor : Dr L.G.Johnson                    */
/*                                                           */
/*         ----------------------------------------          */
/*                                                           */
/*                                                           */
/*      METASM is a meta-assembler designed to assemble      */
/*      microprograms of wordlengths varying from 1 to       */
/*      256 bits. The program has been written in C          */
/*      language & is highly portable. The program can       */
/*      be executed both in MS-DOS & UNIX environments.      */
/*      METASM has been designed to be a two pass ass-       */
/*      embler. The program is capable of finding the        */
/*      micro word length on its own. The program reads      */
/*      a text file containing the program to be assem-      */
/*      bled & then proceeds to assemble the microprogram    */
/*      and stores the assembled data in an output file      */
/*      specified by the user. The program generates a       */
/*      listing with 2 options viz. '-l' and '-ld'. With     */
/*      the '-l' option an ordinary listing is generated     */
/*     With the '-ld' option a complete trace of the         */
/*      assembler can be obtained. The listing is stored     */
/*     in a file called "metasm.lst". The program reserves   */
/*     20 Kbytes of memory for the tables. The program       */
/*      calls 6 main procedures. The microword is sliced     */
/*     into 8 bit slices and these slices are stored in      */
/*     separate files.                                       */
/*************************************************************/


#include <stdio.h>
#define filenam1 "metasm.lst"

/*************************************************************/
/*     The following variables are all global variables.     */
/*     The variable address is the address counter, form     */
/*     keeps count of the format statement, wordlength       */
/*     keeps track of the word length in bytes, eof keeps    */
```

```
/*      track of end of file.                                         */
/**********************************************************************/

int w,i,j,k,m,n,address=0,qq,jj,temp2,addrgen,daf,trueq,debug=1;
int pp,tt,temp3,temp4,y,l,fd,null=0,field,x,form,list=1,symbol=0;
int a,wordlength,eof=100,bitlen=0,true,isnum=1,ismem=1,temp1;
char local[100],temp[15],tempq[15],status,loc[100],loct[256];
char pos[3],see[15],c,outnam[256];
float wlength,wordsize;
FILE *fopen(),*fp,*outfile,*lst;

/**********************************************************************/
/*      The structure with name tag addrtable is a data              */
/*      structure which forms a table of the address names           */
/*      and their corresponding addresses allocated by the           */
/*      assembler.                                                   */
/**********************************************************************/

struct addrtable {
        char addrname[7];
        int contents;
};


/********************************************************************/
/*      The structure with name tag symtable is a data             */
/*      structure which forms a symbol table containing            */
/*      information about the name of the symbol, its              */
/*      definition, as defined by the user, length of the          */
/*      symbol and the bit position from where the symbol          */
/*      is located. This table can hold 30 symbols.                */
/********************************************************************/

struct symtable {
        char name[10],def[9];
        int bitlength,position;
};

/********************************************************************/
/*      The structure with name tag formatb is a data              */
/*      structure which forms a table containing the               */
/*      field names in a particular format statement. 40           */
/*      such structures are stored in an array called              */
/*      formatab. This means that a single format statement        */
/*      can contain a maximum of 40 field names. To make           */
/*      room for more field names in a format statement            */
/*      the size of the formatab array has to be increased         */
/********************************************************************/

struct formatb {
        char formnam[8];
        int bitsize,posn;
}formatab[40];
```

```
/****************************************************************/
/*      The following statement is unique to the C language   */
/*      Here a union of a byte over 8 bits of data was         */
/*      desired. This particular data structure allows         */
/*      access to all 8 bits of the word in which case the     */
/*      word can be read as a byte or allows access to         */
/*      individual bits in the word. This allows manipulation*/
/*      of bits in the word. 32 such 8 bit words make a        */
/*      256 bit word.                                          */
/****************************************************************/

union format {
        unsigned char all;
        struct {
                unsigned bit0:1;
                unsigned bit1:1;
                unsigned bit2:1;
                unsigned bit3:1;
                unsigned bit4:1;
                unsigned bit5:1;
                unsigned bit6:1;
                unsigned bit7:1;
        }part;
}flag[32];

/**************************************************************/
/*      The data structure with name tag fieldtable          */
/*      symbol tables. The structure can hold 26 symbol      */
/*      tables. All the elements of a particular symbol      */
/*      table have the first character of their names        */
/*      being a capital letter. The member tab_count keeps */
/*      track of the locations used in the symbol table.    */
/*      The maximum number of symbols in a symbol table      */
/*      is currently restricted to 30. To make room for      */
/*      more symbols the size of the array symtab has to     */
/*      increased from 30 to the desired value.              */
/**************************************************************/

struct fieldtable {
        struct symtable symtab[30];
        int tab_count;
};

/*************************************************************/
/*      The structure forms contains data of type           */
/*      symtable. The array forms contains 5 such           */
/*      structures. Each element of this array              */
/*      corresponds to a format statement. Thus only        */
/*      five format statements are allowed at this          */
/*      time. To make room for more format statements       */
/*      increase the size of the array tab.                 */
/*************************************************************/

struct forms {
```

```
        struct formatb formatab[30];
}tab[5];

/*************************************************************/
/*      The following two data structures are designed     */
/*      to hold the file names and the corresponding       */
/*      file pointers associated with these file names     */
/*      It is in these files that the sliced output        */
/*      is stored.                                          */
/*************************************************************/

struct names {
        char lot[256];
} flname[32];

struct pointers {
        FILE *outp;
}flpoint[32];
struct fieldtable fieldtab[26];
struct symtable symtab[30];
struct addrtable addtab[200];
union format flag[32];
struct part;
struct forms tab[5];

/*************************************************************/
/*                                                          */
/*      The main program first displays the current        */
/*      working directory on the screen and then prompts   */
/*      the user to specify the input file name where      */
/*      the microprogram to be assembled is stored.        */
/*      Then it specifies the user to specify the output   */
/*      file name where the assembled microprogram is to   */
/*      be stored. If the user wants a listing he appends  */
/*      the option command to the output file name.        */
/*                                                          */
/*      Procedures called:      1. init()                  */
/*                              2. creat_table()           */
/*                              3. chk_ovrlp()             */
/*                              4. getfnam()               */
/*                              5. pass1()                 */
/*                              6. pass2()                 */
/*                                                          */
/*      integer variables used: i.                          */
/*                                                          */
/*      string variables used: outnam[256],loct[256].      */
/*                                                          */
/*************************************************************/

main()
{
        int i;
        system("ls -x");                /* print working directory */
        printf("0);
```

```
printf("0);
printf("Specify input filename :");
scanf("%s",loct);
printf("0);
printf("0);
printf("Output filename :");
scanf("%s",outnam);
printf("0);
i=0;
while(outnam[i]!=' ') {
        i++;
}
i--;
if(outnam[i]=='d') {    /* if '-ld' option then set debug=0 */
        debug=0;        /* and list=0.                      */
        i--;
        if(outnam[i]=='l') {
                i--;
                if(outnam[i]=='-')
                        list=0;
                outnam[i]=' ';
        }
}
if(outnam[i]=='l') {    /* if '-l' option the list=0. */
        i--;
        if(outnam[i]=='-')
                list=0;
        outnam[i]=' ';
}
if(list==0){
        lst=fopen(filenam1,"w+");
        fprintf(lst,"        This meta-assembler has been designed by0);
        fprintf(lst,"                    SHARAD  MURTHY          0);
        fprintf(lst,"                  under the  guidance of     0);
        fprintf(lst,"                   Dr. L. G. JOHNSON         0);
        fprintf(lst,"   M E T A - A S S E M B L E R    L I S T I N G0);
        fprintf(lst,"   ---------------------------------------------0);
        fprintf(lst,"0);
        fprintf(lst,"INPUT FILENAME :%s0,loct);
        fprintf(lst,"OUTPUT FILENAME :%s0,outnam);
}
init();         /* call initialize procedure. */

creat_table();  /* call procedure to fill tables. */

chk_ovrlp();    /* call procedure to check for overlapping */
                /* bit fields.                              */
pass1();        /* call the first pass of the assembler */

getfnam(wordlength);    /* call proc to get file names where */
                        /* sliced output is to be stored. */

pass2();        /* call pass2 of the assembler */
```

```
}

/**********************************************************/
/*                                                        */
/*     This proc initializes all the tables in the        */
/*     program containing names. The initialization       */
/*     process consists of writing the pattern "eos"      */
/*     in all the locations of the tables. All bits of    */
/*     the microword are set to zero.                     */
/*                                                        */
/*     procedures called: None.                           */
/*                                                        */
/*     integer variables used: jj,i,qq,m,address.         */
/*                                                        */
/*     string variables used: None.                       */
/*                                                        */
/**********************************************************/

init()
{

        /* initialize field table */

        printf("Please wait!creation of tables in progress0);
        for(jj=0;jj<26;jj++) {
                for(qq=0;qq<30;qq++) {
                        i=0;
                        fieldtab[jj].symtab[qq].name[i]='e';
                        i++;
                        fieldtab[jj].symtab[qq].name[i]='o';
                        i++;
                        fieldtab[jj].symtab[qq].name[i]='s';
                        i++;
                        fieldtab[jj].symtab[qq].name[i]=' ';
                }
                fieldtab[jj].tab_count=0;
        }

        /* initialize address table */

        for(jj=0;jj<200;jj++) {
                i=0;
                addtab[jj].addrname[i]='e';
                i++;
                addtab[jj].addrname[i]='o';
                i++;
                addtab[jj].addrname[i]='s';
                i++;
                addtab[jj].addrname[i]=' ';
        }

        /* initialize format table */
```

```
        for(qq=0;qq<5;qq++) {
                for(jj=0;jj<40;jj++) {
                        i=0;
                        tab[qq].formatab[jj].formnam[i]='e';
                        i++;
                        tab[qq].formatab[jj].formnam[i]='o';
                        i++;
                        tab[qq].formatab[jj].formnam[i]='s';
                        i++;
                        tab[qq].formatab[jj].formnam[i]=' ';
                }
        }

        /* set all bits of microword to zero */

        for(m=0;m<32;m++) {
                flag[m].all=0x00;
        }
        /* set address counter to zero. */

        address=0x00;
}


/*****************************************************************/
/*                                                               */
/*      The following procedure reads the format state-          */
/*      ments and enters all the field names in the              */
/*      format table corresponding to the format state-          */
/*      ment. Next all the symbols in the field define           */
/*      statements are read and their names and definition       */
/*       are copied into the the field table along with          */
/*      their size and position of origin in the micro           */
/*      word. In this process the wordlength is also             */
/*      determined.                                              */
/*                                                               */
/*      procedures called :        1. skip()                     */
/*                                 2. leave()                    */
/*                                 3. member()                   */
/*                                 4. is_number()                */
/*                                 5. chk_ovfw()                 */
/*                                                               */
/*      integer variables used : qq,i,j,k,temp3,temp4,           */
/*                                field,true,wordlength,         */
/*                                list,debug,form.               */
/*                                                               */
/*      floating point variables : wordsize and wlength          */
/*                                                               */
/*      string variables used: loc[100], temp[15],               */
/*                             pos[3], loct[256],                */
/*                             see[15].                          */
/*                                                               */
/*****************************************************************/
```

```
creat_table()
{

        fp=fopen(loct,"r");      /* open input file */
        if(fp==0) {
                printf("Error: %s file not found",loct);
                exit(1);
        }
        eof=fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        if(debug==0)
                fprintf(lst,"%s0,loc);

        i=0,form=wordlength=0;
        true=1;
        qq=form=j=i=0;
        skip();
        /* check for psuedo instruction format */
        if(strncmp(loc,"FORMAT",6)!=null) {
                printf("error:FORMAT statement missing0);
                exit(1);
        }

        /* if psuedo instruction 'FORMAT' present copy all */
        /* field names into format table */

        while(strncmp(loc,"FORMAT",6)==null) {
                true=1,i=6;
                if(loc[i]!=' ') {
                        printf("error:Type space after FORMAT0);
                }
                if(loc[i]==' ')
                        i++;
                while(true != 0) {
                        if(eof==0)
                                break;
                        while((loc[i]!='0)&&(loc[i]!=';')) {
                                if(eof==0)
                                        break;
                                while(loc[i]==' ')
                                        i++;
                                while((loc[i]!=',')&&(loc[i]!=';')) {
                                        if((loc[i]==' ')||(loc[i]=='0'))
                                                break;
                                        temp[j]=loc[i];
                                        j++,i++;
                                        if(j > 8) {
                                                printf("error : field name too long in0);
                                                printf("%s0,loc);
                                                exit();
                                        }
                                }
                                while(loc[i]==' ')
```

```
                        i++;
                if((loc[i]!=',')||(loc[i]!=';')) {
                        if(loc[i]=='0) {
                                fgets(loc,100,fp);
                                while(strlen(loc)==1)
                                        fgets(loc,100,fp);
                                skip();
                                i=0;
                                if(loc[i]!=',')
                                        leave(',',loc);
                        }
                }
                else {
                        printf("error: ',' operator missing in %s0,loc);
                        exit();
                }
                temp[j]=' '; /* array temp contains */
                                /* field name */

/* check if the field name contains any reserved symbols */

                ismem=chk_rsvd_sym(temp);
        /* if yes then ismem=0. */

                if(ismem==0) {
                        if(list==0) {
                                fprintf(lst,"error: reserved symbol in field name in0);
                                fprintf(lst,"%s0,loc);
                        }
                        printf("error: reserved symbol in field name in 0);
                        printf("%s0,loc);
                        exit(1);
                }
                jj=0;
/* check if field name is redefined */

                while(strncmp(tab[form].formatab[jj].formnam,"eos",3) != null() {
                        if(strcmp(tab[form].formatab[jj].formnam,temp)==null() {
                                if(list==0) {
                                        fprintf(lst,"error: field %s redefined in 0,temp);
                                        fprintf(lst,"%s0,loc);
                                }
                                printf("error: field %s redefined in 0,temp);
                                printf("%s0,loc);
                                exit();
                        }
                        jj++;
                }

                if(debug==0)
                        fprintf(lst,"%s0,temp);
                strcpy(tab[form].formatab[qq].formnam,temp);
                if(loc[i]==';')
                        break;
```

```
                        i++,qq++,j=0;
                }
                if(loc[i]==';')
                        true=0;
                eof=fgets(loc,100,fp);
                while(strlen(loc)==1)     {
                        fgets(loc,100,fp);
                }
                skip();
                if(debug==0)
                        fprintf(lst,"%s0,loc);
                i=j=0;
        }
        qq=0;
        form++;

}
skip();

/* check if psuedo instruction 'DEF' present */
if(strncmp(loc,"DEF",3)!=null) {
        if(list==0) {
                fprintf(lst,"Error:DEF statement missing0);
        }
        else {
                printf("error:DEF statement missing0);
                exit(1);
        }
}
eof=fgets(loc,100,fp);
while(strlen(loc)==1)
        fgets(loc,100,fp);
if(debug==0)
        fprintf(lst,"%s0,loc);
qq=j=0,wordsize=wlength=0.0;
skip();

/* as long as not "ENDEF" copy field symbol name and */
/* definition along with size and position of origin into */
/* the field table.                                   */

while((strncmp(loc,"FIELD",5)==null)&&(strncmp(loc,"ENDEF",5)!=null)){
        i=5,k=0;
        if(loc[i]!=' ') {
                if(list==0) {

                        fprintf(lst,"error:enter a space after FIELD0);
                }
                else {
                        printf("error:enter a space after FIELD0);
                }
        }
        if(loc[i]==' ')
```

```
        i++;
while(loc[i]!=' ') {
        if((loc[i]==' ')||(loc[i]=='0'))/*check to*/
                /* see if line overflows */
                break;
        see[k]=loc[i];
        k++,i++;
        if(k>8) {
                if(list==0) {
                        fprintf(lst,"error: fieldname more than 8 characters long0);
                }
                else {
                        printf("error: fieldname more than 8 characters long0);
                        exit(1);
                }
        }
}
while(loc[i]==' ')
        i++;
if(loc[i]=='0') {
        fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        skip();
        i=0;
        if((loc[i]==',')||(loc[i]==';'))
                leave(',',loc);
        if((loc[i]=='=')||(loc[i]==':'))
                leave('=',loc);
}
see[k]=' ';
k=0;
while((loc[i]!=',')&&(loc[i]!=';')) {
        if((loc[i]==' ')||(loc[i]=='0'))
                break;
        pos[k]=loc[i];
        k++,i++;
        if( k > 2) {
                printf("error: ',' separator missing or number > 2560);
                printf(" in %s0,loc);
                exit();
        }
}
while(loc[i]==' ')
        i++;
if((loc[i]!=',')||(loc[i]!=';'))
        if(loc[i]=='0') {
                fgets(loc,100,fp);
                while(strlen(loc)==1)
                        fgets(loc,100,fp);
                skip();
                i=0;
                if(loc[i]!=',')
                        leave(',',loc);
```

```
        };

pos[k]=' ';
if(loc[i]==',')
        true=1;
i++;
while(loc[i]==' ')
        i++;
if(loc[i]=='0) {
        fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        skip();
        i=0;
}
temp4=atoi(pos); /* temp4 contains position of most */
                 /* significant bit of the symbol */
                 /* in the microword              */

if((temp4 > 256)||(temp4 < 0)) {
        printf("error: bit position out of range in 0);
        printf("%s0,loc);
}
k=0;
while((loc[i]!=';') && (true==1)) {
        if((loc[i]==' ')||(loc[i]=='0))
                break;
        pos[k]=loc[i];
        k++,i++;
        if(k>2) {
                printf("error : ',' separator missing or number > 256 in0);
                printf("%s0);
                exit();
        }
}
while(loc[i]==' ')
        i++;
if(loc[i]!=';')
        if(loc[i]=='0) {
                fgets(loc,100,fp);
                while(strlen(loc)==1)
                        fgets(loc,100,fp);
                skip();
                i=0;
                if(loc[i]!=';')
                        leave(';',loc);
        };

pos[k]=' ';
true=0;
temp3=atoi(pos);  /* temp3 contains the position of */
                  /* the least significant bit of the */
                  /* symbol in the microword.        */
```

```
	if((temp3 > 256)||(temp3 < 0)) {
		printf("error: bit position out of range0);
		printf(" in %s0,loc);
		exit();
	}
	field=temp4-temp3;	/* field contains the size of */
				/* the symbol */
	field++;
	j=qq=true=0;
	trueq=1;

/* enter this information in the format table also */
	while(trueq!=0) {
		while(strcmp(see,tab[j].formatab[qq].formnam)!=0) {
			if(strncmp(tab[j].formatab[qq].formnam,"eos",3)==null)
				break;
			else
				qq++;
		}
		tab[j].formatab[qq].bitsize=field;
		tab[j].formatab[qq].posn=temp3;
		j++,qq=0;
		if(j> (form-1))
			trueq=0;
	}
	eof=fgets(loc,100,fp);
	while(strlen(loc)==1)
		fgets(loc,100,fp);
	skip();
	if(debug==0)
		fprintf(lst,"%s0,loc);
	l=k=i=0;
	while((strncmp(loc,"FIELD",5)!=null)&&(strncmp(loc,"ENDEF",5)!=null)){
		if(loc[i]<0x41 || loc[i] > 0x5a) {
			if(list==0) {
				fprintf(lst,"serious error0);
				fprintf(lst,"error:Symbol name does not begin with capital letter0);
				fprintf(lst,"Symbol:%s0,loc);
				fclose(lst);
			}
			printf("error:Symbol name does not begin with capital letter0);
			printf("Symbol:%s0,loc);
			exit(1);
		}
		j=loc[i]&0x1f; /* get the hash value to hash */
				/* into the field table */
		chk_ovfw(j);
		true=1;
		while(loc[i] != '=') {
			if(loc[i]==' ') {
				while(loc[i]==' ')
					i++;
				if(loc[i]!='=')
					leave('=',loc);
```

```
                        i--;
                        true=0;
                }
                see[l]=loc[i];
                i++,l++;
                if( l > 9) {
                        printf("error: symbol name more than 9 characters 0);
                        printf("in %s0,loc);
                        exit();
                }

        }
        if(true==0)
                l--;
        see[l]=' ';
        ismem=chk_rsvd_sym(see);
        if(ismem==0) {
                if(list==0) {
                        fprintf(lst,"error: reserved symbol in symbol name in0);
                        fprintf(lst,"%s0,loc);
                }
                printf("error: reserved symbol in symbol name in0);
                printf("%s0,loc);
                exit();
        }
        k=0,l=0;
        while(strncmp(fieldtab[j].symtab[k].name,"eos",3)!=null){
                if(strcmp(fieldtab[j].symtab[k].name,see)==null) {
                        if(list==0) {
                                fprintf(lst,"error : symbol %s redefined in 0,see);
                                fprintf(lst,"%s0,loc);
                        }
                        printf("error: symbol %s redefined in 0,see);
                        printf("%s0,loc);
                        exit();
                }
                k++;
        }
        fieldtab[j].symtab[k].name[l]=see[l];
        l=0;
        while(see[l]!=' ') {
                fieldtab[j].symtab[k].name[l]=see[l];
                l++;
        }
        fieldtab[j].symtab[k].name[l]=' ';
        l=0,i++;
        while(loc[i] != ';'){
                while(loc[i]==' ')
                        i++;
                if(loc[i]=='0) {
                        fgets(loc,100,fp);
                        while(strlen(loc)==1)
                                fgets(loc,100,fp);
                        skip();
```

```
                                        i=0;
                                        if(loc[i]!=';')
                                                leave(';',loc);
                                }
                                if(loc[i]==';')
                                        break;
                                if(loc[i] < '0' || loc[i] > '1') {
                                        printf("error: Bit value other than 1 or 0 in0);
                                        printf("%s0,loc);
                                        exit(1);
                                }
                                fieldtab[j].symtab[k].def[l]=loc[i];
                                i++,l++;
                                if(l > 9) {
                                        printf("error: can not handle symbols with more than 9 bits0);
                                        printf ("%s0,loc);
                                        exit();
                                }
                        }
                        fieldtab[j].symtab[k].def[l]=' ';
                        fieldtab[j].symtab[k].bitlength=field;

                        fieldtab[j].symtab[k].position=temp3;

                        eof=fgets(loc,100,fp);
                        while(strlen(loc)==1)
                                fgets(loc,100,fp);
                        skip();
                        if(debug==0)
                                fprintf(lst,"%s0,loc);
                        l=i=k=0;
                }
                temp3=temp4=0;
        }
        qq=wordlength=0;
        /* compute microword length */
        while(strncmp(tab[0].formatab[qq].formnam,"eos",3)!=null) {
                wordlength=wordlength+tab[0].formatab[qq].bitsize;
                qq++;
        }
        wordsize=wordlength;
        wlength=wordsize/8.0;
        wordlength=wlength;
        wordsize=wlength- wordlength;
        if(wordsize>0.0)
                wordlength=wordlength+1;
        if(list==0)
                fprintf(lst,"wordlength=%d0,wordlength);
        fclose(fp);
}

/*******************************************************************/
/*                                                                 */
```

```
/*      The following procedure checks for overlapping bitfields */
/*      in the same format statement. If an overlap is found     */
/*      then an error message is printed on the screen and the   */
/*      program is aborted. Else it prints an affirmative        */
/*      message.                                                 */
/*                                                               */
/*      Procedures called : None.                                */
/*                                                               */
/*      integer variables used: w,j,qq,m,n,temp2,temp3,temp4,x   */
/*                              tt,true,trueq,form,daf.          */
/*                                                               */
/*      string variables used : temp[15].                        */
/*                                                               */
/*****************************************************************/

chk_ovrlp()
{

        if(list==0)
                fprintf(lst,"Please wait,checking for overlappingbitfields0);
        printf("Please wait,checking for overlapping bit fields0);
        w=j=qq=0,true=trueq=1;

        while(true!=0) {
                while(trueq!=0) {

                        if(strncmp(tab[j].formatab[w].formnam,"eos",3)!=0) {
                                strcpy(temp,tab[j].formatab[w].formnam);

                                temp3=tab[j].formatab[w].posn;
                                temp4=tab[j].formatab[w].bitsize;
                                daf=temp3+(temp4-1);

                        }
                        if(strncmp(tab[j].formatab[w].formnam,"eos",3)==0)


                                trueq=0;
                        qq=w;
                        qq++;
                        while(strncmp(tab[j].formatab[qq].formnam,"eos",3)!=0) {

                                temp2=tab[j].formatab[qq].posn;
                                x=tab[j].formatab[qq].bitsize;
                                tt=temp2+(x-1);
                                n=temp2;
                                m=temp3;
                                if((m==n)||((n>m)&&(n<=daf))||((n<m)&&(m<=tt))) {

                                        if(list==0) {
                                                fprintf(lst,"error:You have overlapping bit fields in your definition0);
                                                fprintf(lst,"FIELD:%s overlaps FIELD:%s0,temp,tab[j].formatab[qq].formnam);
                                                fclose(lst);
                                        }
```

```
                                        printf("error:You have overlapping bit fields in your definition0);
                                        printf("FIELD:%s is overlapping FIELD:%s0,temp,tab[j].formatab[qq].formnam);
                                        printf("Please correct the error & try to assemble again0);
                                        exit(1);
                                }
                                qq++;

                        }
                        if(trueq==0)
                                break;
                        w++;
                }
                if(j>(form-1))

                        true=0;
                j++;
                w=0,trueq=1;
        }

        printf("Good, you have no overlapping bit fields0);
}


/****************************************************************/
/*                                                              */
/*      The following procedure is the first pass of the        */
/*      assembler. The microprogram is scanned from start       */
/*      to end and as and when address labels are encountered  */
/*      an absolute value is assigned to the address label.    */
/*      An address counter keeps count of the end of micro-    */
/*      instruction character which happens to be ':'. Both   */
/*      the address label and its corresponding absolute       */
/*      value are stored in the address table for future       */
/*      references.                                            */
/*                                                              */
/*      procedures called: None.                               */
/*                                                              */
/*      integer variables used: j,qq,i,x,address,debug,        */
/*                              true,trueq,eof,status.         */
/*                                                              */
/*      string variables used: loc[100],loct[256].            */
/*                                                              */
/****************************************************************/

pass1()

{

        fp=fopen(loct,"r");
        fgets(loc,100,fp);
        while(strlen(loc)==1)
```

```
        fgets(loc,100,fp);
skip();
if(debug==0)
        fprintf(lst,"%s0,loc);
while(strncmp(loc,"ENDEF",5)!=null) {
        if(eof==0) {
                if(list==0) {
                        fprintf(lst,"error: ENDEF statement missing0);
                }
                printf("error: ENDEF statement missing0);
                exit();
        }
        eof=fgets(loc,100,fp);
        while(strlen(loc)==1)
                eof=fgets(loc,100,fp);
        skip();
}
true=1;
address=0x00;


eof=fgets(loc,100,fp);
while(strlen(loc)==1)
        fgets(loc,100,fp);
skip();
if(debug==0)
        fprintf(lst,"%s0,loc);
j=qq=i=x=0;
while(strncmp(loc,"END",3)!=0) {
        if(eof==0) {
                if(list==0){

                        fprintf(lst,"Error:END statement missing0);
                }
                printf("Error: END statement missing0);

        }
        if(eof==0)
                break;
        trueq=0;
        j=0;
        while(loc[i]!=':') {
                if((loc[i]==' ')||(loc[i]=='0'))
                        break;
                if((i>10)&&(loc[i]!=':'))
                        break;
                temp[j]=loc[i];
                j++,i++;
        }
        while(loc[i]==' ')
                i++;
        if(loc[i]=='0') {
                fgets(loc,100,fp);
                while(strlen(loc)==1)
```

```
                        fgets(loc,100,fp);
                skip();
                i=0;
        }
        temp[j]=' ';
        j=0;
        if(loc[i]!=':')
                trueq=1;
        if(trueq==0) {
                while(temp[j] != ' ') {
                        addtab[qq].addrname[j]=temp[j];
                        if(j>6) {
                                if(list==0) {
                                        fprintf(lst,"error:address label more than 6 characters long0);
                                        fclose(lst);
                                }
                                printf("error:address label more than 6 characters long0);
                                exit(1);
                        }
                        j++,i++;
                }
                addtab[qq].addrname[j]=' ';
                addtab[qq].contents=address;


                if(debug==0)
                        fprintf(lst,"     %s = %x0,addtab[qq].addrname,addtab[qq].contents);
                qq++;
        }
        while(true != 0) {
                if(eof==0)
                        break;
                while((loc[i]!='0)&&(loc[i]!=';')) {
                        i++;
                }
                if(loc[i]==';') {
                        true=0;
                        address++;
                }
                eof=fgets(loc,100,fp);
                while(strlen(loc)==1)
                        fgets(loc,100,fp);
                skip();
                if(debug==0)
                        fprintf(lst,"%s0,loc);
                j=i=0;
        }
        true=1;
}
status=fclose(fp);
if(status!=0) {
        printf("unsuccessful file closure:%s:I/O error0,loct);
}
j=0;
```

```
        fclose(fp);
}

/****************************************************************/
/*                                                              */
/*      This procedure is the second pass of the assembler     */
/*      In this procedure the microprogram is actually         */
/*      assembled and stored in a separate file. For           */
/*      assembling this procedure refers to the field table    */
/*      the address table and the format table. For sear-      */
/*      ching the format and the address table, linear         */
/*      search technique has been adopted. For searching       */
/*      the field table, a technique similar to hashing        */
/*      has been used. The microprogram is scanned             */
/*      microinstruction by microinstruction till END          */
/*      statement or end of file is encountered. For every     */
/*      microinstruction, the field symbols are read one       */
/*      after the other and a check is made of the field       */
/*      table and when a match is found, the field symbol      */
/*      size and the position of origin of the                 */
/*      in the microword are read into a buffer. Next the      */
/*      value of the field symbol is read and the value        */
/*      is examined bi by bit and the field symbol is asse-    */
/*      mleb by setting the the appropriate bits in the        */
/*      microword. After the microword has been assembled      */
/*      completely, it sliced into 8 bit slices and these      */
/*      slices are stored in files whose names are obtained    */
/*      from the user interactively ibefore this procedure.    */
/*                                                              */
/*      procedures called:      1. stat()                       */
/*                              2. dump()                       */
/*                              3. member()                     */
/*                              4. is_number()                  */
/*                              5. chk_rsvd_sym()               */
/*                              6. leave()                      */
/*                                                              */
/*      integer variables used: i,j,k,l,m,n,jj,tt,y,pp,x,       */
/*                              temp1,temp2,temp3,temp4,        */
/*                              address,eof,list,debug,         */
/*                              symbol, wlength, daf,trueq      */
/*                              addrgen,ismem,isnum,true.       */
/*                                                              */
/*      string variables used: temp[15], see[15], pos[3],       */
/*                             loc[100].                        */
/*                                                              */
/****************************************************************/


pass2()

{
        address=0x00;
```

```
outfile=fopen(outnam,"w+");

fp=fopen(loct,"r");
eof=fgets(loc,100,fp);/* if eof=0 then end of file encountered*/
while(strlen(loc)==1)
        fgets(loc,100,fp);
skip();
if(list==0) {
        fprintf(lst,"PASS-1 complete 0);
}
while(strncmp(loc,"ENDEF",5)!=null) {
        eof=fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        skip();
}
i=k=m=n=0;
eof=fgets(loc,100,fp);
while(strlen(loc)==1)
        fgets(loc,100,fp);
skip();
if(list==0) {
        fprintf(lst,"PASS-2 BEGINS ===> Assembly in progress0);
        fprintf(lst,"1s",loc);
}
i=0;
while(loc[i]!=':') {
        if((loc[i]==' ')!!(loc[i]=='0))
                break;
        if((i>5)&&(loc[i]!=':'))
                break;
        i++;
}
while(loc[i]==' ')
        i++;
if(loc[i]=='0){
        fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        skip();
        i=0;
}

if(loc[i]!=':')
        i=0;
else
        i++;
daf=0,addrgen=1;/* if daf=0 then field symbol to be */
                /* assembled, if addrgen=1 then value */
                /* to be assembled is either a decimal*/
                /* number or an address label. */

while(loc[i]==' ')
        i++;
```

```
/* strart reading microinstructions */
while(strncmp(loc,"END",3)!=null) {
        wlength=0;
        if(eof==0)
                break;
        while(loc[i]==' ')
                i++;
        while((loc[i]!='0)&&(loc[i]!=';')) {
                n=0;
                if(eof==0)
                        break;
                while(loc[i]==' ')
                        i++;
                while((loc[i]!=',')&&(loc[i]!=';')) {
                        if(eof==0)
                                break;
                        if((loc[i]==' ')||(loc[i]=='0))
                                break;
                        temp[n]=loc[i];
                        n++,i++;
                }
                while(loc[i]==' ')
                        i++;
                if(loc[i]=='0)  {
                        fgets(loc,100,fp);
                        while(strlen(loc)==1)
                                fgets(loc,100,fp);
                        skip();
                        i=0;
                }
                temp[n]=' '; /* read all characters up to */
                        /* ',' chatacter into array temp */
                ismem=member('=',temp); /* check if symbol */
                                /* '=' present in temp */
                if((n>9)&&(ismem!=0)) {
                        if(list==0) {
                                fprintf(lst,"error: symbol name more than 9 characters in 0);
                                fprintf(lst,"%s0,loc);
                                fclose(lst);
                        }
                        printf("error: symbol name more than 9 characters in 0);
                        printf("%s0);
                        exit();
                }
                n=0;
                qq=jj=n=0;

    /* if '=' symbol present then read all characters to the */
    /* left of '=' symbol in to an array see */

                if(ismem==0) {
                        n=0;
                        while(temp[n]!='=') {
                                see[n]=temp[n];
```

```
                        n++;
                }
                see[n]=' ';
                w=0,n++;
                while(temp[n]!=' ') {
                        tempq[w]=temp[n];
                        w++,n++;
                }
                tempq[w]=' '; /* read all characters */
                        /* to the right of '=' symbol */
                        /* into array tempq.    */

/* check if the field name in see is a valid field name. */
/* by checking the format table. */

                for(j=0;j<form;j++) {
                        k=0;
                        while(strncmp(tab[j].formatab[k].formnam,"eos",3)!=null) {
                                if(strcmp(tab[j].formatab[k].formnam,see)==null) {
                                        temp2=tab[j].formatab[k].bitsize;
                                        temp3=tab[j].formatab[k].posn;
                                        true=0;
                                }
                                if(true==0)
                                        break;
                                k++;
                        }
                        if(true==0)
                                break;
                }
                if(true==1) {
                        if(list==0) {
                                fprintf(lst,"error: field %s not found0,see);
                                fclose(lst);
                        }
                        printf("error: field %s not found0,see);
                        exit();
                }
                true=1;
                isnum=is_number(tempq);

/* check if the contents of array tempq is a positive integer */
/* if number then set isnum=0    */

                if(isnum==0) {
                        temp4=atoi(tempq);
                        if(temp4 > 32000) {
                                if(list==0) {
                                        fprintf(lst, "error: value > 32000 in0);
                                        fprintf(lst,"%s0,loc);
                                        fclose(lst);
                                }
                                printf("error: value > 32000 in0);
                                printf("%s0,loc);
```

```
                                        exit();
                                }
                                addrgen=0,daf=1;
                        }

/* if not number check if the contents of tempq */
/* is a field symbol by checking the field table */
                        else {
                                qq=tempq[0]&0x1f;
                                jj=0,symbol=0;
                                while(jj < 30) {
                                        if(strncmp(fieldtab[qq].symtab[jj].name,"eos",3)==null)
                                                symbol=1;
                                        if(symbol==1)
                                                break;
                                        if(strcmp(fieldtab[qq].symtab[jj].name,tempq)==null)
                                                break;
                                        jj++;
                                }
                                if(symbol==0) {
                                        temp1=fieldtab[qq].symtab[jj].bitlength;
                                        if(temp1!=temp2) {
                                                if(list==0) {
                                                        fprintf(lst,"error:Field size does not match field symbol si
                                                        fprintf(lst,"Field :%s0,see);
                                                        fprintf(lst,"Field symbol : %s0,tempq);
                                                }
                                                printf("error:Field size does not match field symbol size0);
                                                printf("Field : %s0,see);
                                                printf("Field symbol :%s0,tempq);
                                                printf("in %s0,loc);
                                                exit();
                                        }
                                        addrgen=1,daf=0;
                                }
                        }

/* if the contents of tempq is neither a number nor */
/* a field symbol check if it is an address label */
/* by checking the address table.              */

                        if((symbol==1)&&(isnum!=0)) {
                                qq=0;

                                while(strcmp(addtab[qq].addrname,tempq)!=null) {
                                        if(strncmp(addtab[qq].addrname,"eos",3)==null) {


                                                if(list==0){
                                                        fprintf(lst,"error:addresslabel %s not found.0,tempq);
                                                        fclose(fp);
                                                }
                                                printf("error:addresslabel %s not found.0,tempq);
                                                exit();
```

```
                                        }
                                        qq++;
                                }
                                temp4=addtab[qq].contents;
                                addrgen=0,daf=1;

                        }
                        true=1;

                }
/* if the array temp does not contain the character '=' */
/* that means temp contains field symbol. Check field */
/* table and set daf=0 */
                else {
                        qq=jj=0;
                        n=0;
                        qq=temp[n]&0x1f;

                        while(jj < 30) {
                                if(strncmp(fieldtab[qq].symtab[jj].name,"eos",3)==null) {

                                        if(list==0) {
                                                fprintf(lst,"error:symbol not found0);
                                                fprintf(lst,"symbol:%s in0,temp);
                                                fprintf(lst,"%s0,loc);
                                                fclose(lst);
                                        }
                                        printf("error:symbol not found0);
                                        printf("Symbol:%s in0,temp);
                                        printf("%s0,loc);
                                        exit(1);
                                }
                                if(strcmp(fieldtab[qq].symtab[jj].name,temp)==null) {

                                        true=0;


                                }
                                if(true==0)
                                        break;
                                jj++;
                        }
                        true=1;


                        temp2=fieldtab[qq].symtab[jj].bitlength;


                        temp3=fieldtab[qq].symtab[jj].position;
                }
                tt=0;

/* locate the position of  origin of the field symbol */
/* in the byte and the byte where the symbol originates */
```

```
/*  temp3 contains the position of origin and tt the  */
/* byte number.                    */

                if((temp3 > 7)&&(temp3<=15)) {
                        temp3= temp3-8;
                        tt=1;
                }
                if((temp3 > 15)&&(temp3 <=23)) {
                        temp3= temp3-16;
                        tt=2;
                }
                if((temp3 > 23)&&(temp3 <= 31)) {
                        temp3=temp3-24;
                        tt=3;
                }
                if((temp3 > 31)&&(temp3 <= 39)) {
                        temp3=temp3-32;
                        tt=4;
                }
                if((temp3 > 39)&&(temp3 <=47)) {
                        temp3=temp3-40;
                        tt=5;
                }
                if((temp3 > 47)&&(temp3 <= 55)) {
                        temp3=temp3-48;
                        tt=6;
                }
                if((temp3 > 55)&&(temp3 <= 63)) {
                        temp3=temp3-56;
                        tt=7;
                }
                if((temp3 > 63)&&(temp3 <= 71)) {
                        temp3=temp3-64;
                        tt=8;
                }
                if((temp3 > 71)&&(temp3 <= 79)) {
                        temp3=temp3-72;
                        tt=9;
                }
                if((temp3 > 79)&&(temp3 <=87)) {
                        temp3=temp3-80;
                        tt=10;
                }
                if((temp3 > 87)&&(temp3 <= 95)) {
                        temp3=temp3-88;
                        tt=11;
                }
                if((temp3 > 95)&&(temp3 <= 103)) {
                        temp3=temp3-96;
                        tt=12;
                }
                if((temp3 > 103)&&(temp3 <= 111)) {
                        temp3=temp3-104;
                        tt=13;
```

```
}
if((temp3 > 111)&&(temp3 <= 119)) {
        temp3=temp3-112;
        tt=14;
}
if((temp3 > 119)&&(temp3 <= 127)) {
        temp3=temp3-120;
        tt=15;
}
if((temp3 > 127)&&(temp3 <=135)) {
        temp3=temp3-128;
        tt=16;
}
if((temp3 > 135)&&(temp3 <=143)) {
        temp3=temp3-136;
        tt=17;
}
if((temp3 > 143)&&(temp3 <=151)) {
        temp3=temp3-144;
        tt=18;
}
if((temp3 > 151)&&(temp3 <= 159)) {
        temp3=temp3-152;
        tt=19;
}
if((temp3 > 159)&&(temp3 <= 167)) {
        temp3=temp3-160;
        tt=20;
}
if((temp3 > 167)&&(temp3 <= 175)) {
        temp3=temp3-168;
        tt=21;
}
if((temp3 > 175)&&(temp3 <= 183)) {
        temp3=temp3-176;
        tt=22;
}
if((temp3 > 183)&&(temp3 <= 191)) {
        temp3=temp3-184;
        tt=23;
}
if((temp3 > 191)&&(temp3 <= 199)) {
        temp3=temp3-192;
        tt=24;
}
if((temp3 > 199)&&(temp3 <=207)) {
        temp3=temp3-200;
        tt=25;
}
if((temp3 > 207)&&(temp3 <= 215)) {
        temp3=temp3-208;
        tt=26;
}
if((temp3 >215)&&(temp3 <= 223)) {
```

```
                        temp3=temp3-216;
                        tt=27;
                }
                if((temp3 >223)&&(temp3 <= 231)) {
                        temp3=temp3-224;
                        tt=28;
                }
                if((temp3 > 231)&&(temp3 <= 239)) {
                        temp3=temp3-232;
                        tt=29;
                }
                if((temp3 > 239)&&(temp3 <= 247)) {
                        temp3=temp3-240;
                        tt=30;
                }
                if((temp3 > 247)&&(temp3 <= 255)) {
                        temp3=temp3-248;
                        tt=31;
                }
                y=0;
                y=temp3;
/* start assembling */
                pp=strlen(fieldtab[qq].symtab[jj].def);
                pp--;
                if(daf==0) {
                        while(temp2 > 0) {
                                y=temp3;
                                while(temp3 <8) {
                                        if(fieldtab[qq].symtab[jj].def[pp]=='1') {
                                                switch(y) {
                                                case 0:
                                                        flag[tt].part.bit0=1;
                                                        break;
                                                case 1:
                                                        flag[tt].part.bit1=1;
                                                        break;
                                                case 2:
                                                        flag[tt].part.bit2=1;
                                                        break;
                                                case 3:
                                                        flag[tt].part.bit3=1;
                                                        break;
                                                case 4:
                                                        flag[tt].part.bit4=1;
                                                        break;
                                                case 5:
                                                        flag[tt].part.bit5=1;
                                                        break;
                                                case 6:
                                                        flag[tt].part.bit6=1;
                                                        break;
                                                case 7:
                                                        flag[tt].part.bit7=1;
                                                        break;
```

```
                                        default:
                                                printf("sharad0);
                                                break;
                                        }
                                }
                                temp2--;
                                if(temp2==0)
                                        break;
                                pp--,temp3++;
                                y=temp3;
                        }
                        temp3=0,tt++;
                }
                daf=1;
        }

/* if address label to be assembled then examine all */
/* the bits of the integer representing address value */

        if(addrgen==0) {
                while(temp2 > 0) {
                        while(temp3 <=8) {

                                switch(x) {
                                case 0:
                                        m=temp4&0x00000001;
                                        break;
                                case 1:
                                        m=temp4&0x00000002;
                                        break;
                                case 2:
                                        m=temp4&0x00000004;
                                        break;
                                case 3:
                                        m=temp4&0x00000008;
                                        break;
                                case 4:
                                        m=temp4&0x00000010;
                                        break;
                                case 5:
                                        m=temp4&0x00000020;
                                        break;
                                case 6:
                                        m=temp4&0x00000040;
                                        break;
                                case 7:
                                        m=temp4&0x00000080;
                                        break;
                                case 8:
                                        m=temp4&0x00000100;
                                        break;
                                case 9:
                                        m=temp4&0x00000200;
```

```
            break;
    case 10:
            m=temp4&0x00000400;
            break;
    case 11:
            m=temp4&0x00000800;
            break;
    case 12:
            m=temp4&0x00001000;
            break;
    case 13:
            m=temp4&0x00002000;
            break;
    case 14:
            m=temp4&0x00004000;
            break;
    case 15:
            m=temp4&0x00008000;
            break;
    case 16:
            m=temp4&0x00010000;
            break;
    case 17:
            m=temp4&0x00020000;
            break;
    case 18:
            m=temp4&0x00040000;
            break;
    case 19:
            m=temp4&0x00080000;
            break;
    case 20:
            m=temp4&0x00100000;
            break;
    case 21:
            m=temp4&0x00200000;
            break;
    case 22:
            m=temp4&0x00400000;
            break;
    case 23:
            m=temp4&0x00800000;
            break;
    case 24:
            m=temp4&0x01000000;
            break;
    case 25:
            m=temp4&0x02000000;
            break;
    case 26:
            m=temp4&0x04000000;
            break;
    case 27:
            m=temp4&0x08000000;
```

```
			break;
		case 28:
			m=temp4&0x10000000;
			break;
		case 29:
			m=temp4&0x20000000;
			break;
		case 30:
			m=temp4&0x40000000;
			break;
		case 31:
			m=temp4&0x80000000;
			break;
		default:
			printf("vasant0);
			break;
		}

		if(m!=0) {
			switch(y) {
			case 0:
				flag[tt].part.bit0=1;
				break;
			case 1:
				flag[tt].part.bit1=1;
				break;
			case 2:
				flag[tt].part.bit2=1;
				break;
			case 3:
				flag[tt].part.bit3=1;
				break;
			case 4:
				flag[tt].part.bit4=1;
				break;
			case 5:
				flag[tt].part.bit5=1;
				break;
			case 6:
				flag[tt].part.bit6=1;
				break;
			case 7:
				flag[tt].part.bit7=1;
				break;
			default:
				printf("rama0);
			}
		}
		temp2--;
		if(temp2==0)
			break;
		pp++,temp3++,x++;
		y=temp3;
	}
```

```
                                temp3=0,tt++;
                        }
                        addrgen=1;
                }
                x=0,daf=0;              /* set daf flag to zero */
                if(loc[i]==';')
                        break;
                i++;
        }
        if(loc[i]==';') {
                if(list==0)
                        fprintf(lst,"%x :",address);
                address++;                                              /* jj is the wordlength in bytes */
                jj=wordlength-1;
                while(jj >= 0) {

                        fprintf(outfile,"%x",flag[jj].all);
                        if(list==0)
                                fprintf(lst," %x",flag[jj].all);
                        dump(flag[jj].all,flname[jj].lot,flpoint[jj].outp);
                        jj--;
                }
                fprintf(outfile,"%c",'0);
                for(m=0;m<32;m++) {
                        flag[m].all=0x00;
                }
        }
        eof=fgets(loc,100,fp);
        while(strlen(loc)==1)
                fgets(loc,100,fp);
        skip();
        if(list==0)
                fprintf(lst,"%s",loc);
        i=0;
        while(loc[i]!=':') {
                if((i>5)&&(loc[i]!=':'))
                        break;
                i++;
        }
        if(loc[i]!=':')
                i=0;
        else
                i++;
        while(loc[i]==' ')
                i++;
}
if(list==0)
        stat();
fclose(outfile);
fclose(fp);
if(list==0)
        fclose(lst);
}
```

```
/*************************************************************/
/*                                                           */
/*      The following procedure checks the space used        */
/*      in the symbol table whose hash value is passed       */
/*      as an argument to this function. If the sapce        */
/*      used is greater than 29 then an error message is     */
/*      flashed indicating an overflow and a statistics      */
/*      of the space used in all the tables is printed       */
/*      on the screen.                                       */
/*                                                           */
/*      Procedures called:      1. print_err(num)            */
/*                                                           */
/*      integers used : count, num.                          */
/*                                                           */
/*      string variables used : None.                        */
/*                                                           */
/*************************************************************/

cnk_ovfw(num)
int num;

{

        int count;

        fieldtab[j].tab_count++;
        count=fieldtab[j].tab_count;
        if(count > 29){
                print_err(num);
                exit(1);
        }
}


/************************************************************/
/*                                                          */
/*      This procedure prints the complete statistics of    */
/*      usage of the tables along with the error message    */
/*      indicating which particular symbol table overflowed*/
/*                                                          */
/*      Procedures called : None                            */
/*                                                          */
/*      Integer variables used: charctr,cnt,xy,sym.         */
/*                                                          */
/*      character variable : character                      */
/*                                                          */
/************************************************************/

print_err(charctr)
int charctr;

{
        char character=0x41;
        int sym=0,cnt=41,xy;
```

```
        character = character + charctr;
        printf("error : you nave used more than 29 symbol names begining");
        printf(" witn the character %c /n",cnaracter);
        printf(" Status of symbol table : /n");
        if(list==0) {
                fprintf(lst,"error : you have used more than 29 symbol names begining");
                fprintf(lst,"witn the character %c /n",character);
                fprintf(lst," Status of symbol table :/n");
        }
        while(sym < 26) {
                printf("Number of symbol names begining with cnaracter");
                printf("%c = %d /n",character,fieldtab[sym].tab_count);
                if(list==0) {
                        fprintf(lst,"Number of symbol names begining with character");
                        fprintf(lst,"%c = %d /n",character,fieldtab[sym].tab_count);
                        fclose(lst);
                }
                sym++;
                character=sym|0xe0;
        }
}

/**********************************************************************/
/*                                                                    */
/*      This procedure writes the character passed to it as           */
/*      an argument into a file whose file name and file pointer      */
/*      are also passed as arguments to this function.                */
/*                                                                    */
/*      procedures called : None.                                     */
/*                                                                    */
/*      integer variables used : None                                 */
/*                                                                    */
/**********************************************************************/

dump(c,fnam,sp)

cnar c,fnam[15];
FILE *sp;
{
        sp=fopen(fnam,"a");
        fputc(c,sp);
        putc('0,sp);
        fclose(sp);
}

/**********************************************************************/
/*                                                                    */
/*      The following procedure prompts the user to specify           */
/*      the filename and gets a string wnich is supposed              */
/*      to be a file name and stores it in the structure called       */
/*      names. The function gets as many file names as is specified*/
/*      by the argument passed to it.                                 */
```

```
/*                                                                      */
/************************************************************************/

getfnam(wlen)
int wlen;
{
        int x=0;
        while(x<=wlen-1) {
                printf(" Specify name of output file %d :",x+1);
                scanf("%s",flname[x].lot);
                x++;
        }
}

/************************************************************************/
/*                                                                      */
/*      The following procedure returns zero if the character           */
/*       passed to it as an argument is a member of the string          */
/*      passed to it as an argument, else returns 1.                    */
/*                                                                      */
/************************************************************************/

member(c,array)
char c,*array;
{
        int i=0,true=1;
        while(array[i]!=' ') {
                if(array[i]=='=')
                        true=0;
                if(true==0)
                        break;
                i++;
        }
        return(true);
}

/************************************************************************/
/*                                                                      */
/*      The following procedure returns zero if all the members         */
/*      of the string passed to it as an argument happen to be          */
/*      between 0 and 9, else returns 1.                                */
/*                                                                      */
/************************************************************************/

is_number(array)
char *array;
{
        int true=0,k=0;
        while(array[k]!=' ') {
                if((array[k] <'0')||(array[k] > '9'))
                        true=1;
```

```
                if(true==1)
                        break;
                k++;
        }
        return(true);
}

/*****************************************************************/
/*                                                               */
/*      This procedure removes all the space characters at       */
/*      the begining of the string loc which is a global variable*/
/*      and which contains the text line in the source file      */
/*      which contains the microprogram to be assembled.         */
/*                                                               */
/*****************************************************************/

skip()
{
        char array[256];
        int i=0,j=0;
        while(loc[i]==' ')
                i++;
        while(loc[i]!=' ') {
                array[j]=loc[i];
                j++,i++;
        }
        array[j]=' ';
        i=j=0;
        while(array[j]!=' ') {
                loc[i]=array[j];
                i++,j++;
        }
        loc[i]=' ';
}

/*****************************************************************/
/*                                                               */
/*      This procedure returns zero if the string passed to      */
/*      it as an argument contains any of the reserved symbols   */
/*      viz. '=', ';', ',', ':'; else returns 1.                 */
/*                                                               */
/*****************************************************************/

chk_rsvd_sym(array)
char *array;
{
        int true=1,i=0;
        char valray[5];
        valray[0]=';';
        valray[1]=',';
        valray[2]=':';
        valray[3]='=';
```

```
        while(i<4) {
                true=member(valray[i],array);
                if(true==0)
                        break;
                i++;
        }
        return(true);
}

/*************************************************************/
/*                                                           */
/*      This procedure prints an error message stating       */
/*      that the character passed to it as an argument       */
/*      is missing from the string passed to it as  an       */
/*      argument and then exits the program.                 */
/*                                                           */
/*************************************************************/

leave(c,array)
char c,*array;
{
        if(list==0)
                fprintf(lst,"error: '%c' operator missing in %s0,c,array);
        printf("error : '%c' operator missing in %s0,c,array);
        exit();
}

/*************************************************************/
/*                                                           */
/*      This procedure prints an error message followed by   */
/*      a complete statistics of the usage of the table      */
/*      in the list file called metasm.lst.                  */
/*                                                           */
/*************************************************************/

stat()
{
        int yy=0,addrcnt=0;
        char character=0x41;

        while(yy < 26) {
                fprintf(lst,"number of symbols starting with character");
                fprintf(lst," %c = %d0,character,fieldtab[yy].tab_count);
                character++;
                addrcnt=addrcnt+ fieldtab[yy].tab_count;
                yy++;
        }
        fprintf(lst,"total number of symbols used = %d0,addrcnt);
}
```

BIBLIOGRAPHY

[1] Samir S Husson (1970). Microprogrammed principles and practices, IBM Systems Research Institute, New York: Prentice Hall, Inc.

[2] John J Donovan (1972). Systems Programming, New York: McGraw Hill.

[3] Robert M Graham (1975). Principles of systems programming, John Willey and Sons, Inc.

[4] Brian W Kernighan and Dennis M Ritchie (1978). The C programming language, Englewood Cliffs, NJ: Prentice Hall.

[5] Borland Turbo C reference manual (1987). Borland Corporation, CA.

VITA

Sharadchandra R. Murthy

Candidate for the degree of

Master of Science

Thesis: DESIGN OF A META-ASSEMBLER

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Mysore, India, December 27, 1962, the son of T.G.R.Murthy and Chaya Murthy.

Education: Received Bachelor of Engineering in Electronics from Bangalore University, India in December, 1984; completed requirements for the Master of Science degree at Oklahoma State University in July, 1988.

Professional Experience: Customer Support Engineer, PSI Data Systems, India, January, 1985 to June, 1986; Research Assistant, Department of Entomology, Oklahoma State University, October 1986, to May, 1987; Teaching Assistant, Department of Electrical and Computer Engineering, Oklahoma State University, August, 1987 to May, 1988.